AFRL-RY-WP-TR-2015-0183

CLOUD INTRUSION DETECTION AND REPAIR (CIDAR)

Stelios Sidiroglou, Jeff Perkins, and Martin Rinard

Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory

FEBRUARY 2016
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the USAF 88th Air Base Wing (88 ABW) Public Affairs Office (PAO) and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2015-0183 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

// Signature//

PHILIP D. MUMFORD
Program Engineer
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

// Signature//

DAVID G. HAGSTROM, Chief
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

// Signature//

TODD A KASTLE, Division Chief
Spectrum Warfare Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2016	Final	30 September 2011 – 30 September 2015

4. TITLE AND SUBTITLE
CLOUD INTRUSION DETECTION AND REPAIR (CIDAR)

5a. CONTRACT NUMBER
FA8650-11-C-7192

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Stelios Sidiroglou, Jeff Perkins, and Martin Rinard

5d. PROJECT NUMBER
3000

5e. TASK NUMBER
YW

5f. WORK UNIT NUMBER
Y0PM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory
32 Vassar St, Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency
(DARPA/I2O)
675 North Randolph Street
Arlington, VA 22203-2114

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYW

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2015-0183

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The U.S. Government is joint author of the work and has the right to use, modify, reproduce, release, perform, display or disclose the work. PAO case number 88ABW-2015-5735, Clearance Date 1 December 2015. Report contains color. The material is based on research sponsored by Air Force Research Laboratory (AFRL) and the Defense Advanced Research Agency (DARPA) under agreement number FA8650-11-C-7192. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and the Defense Advanced Research Agency (DARPA) or the U.S. Government.

14. ABSTRACT
Despite decades of effort, defect triage and correction remains a central concern in software engineering. Indeed, modern software projects contain so many defects, and the cost of correcting defects remains so large, that projects typically ship with a long list of known but uncorrected defects. Consequences of this unfortunate situation include pervasive security vulnerabilities and the diversion of resources that would be better devoted to other, more productive, activities. The goal of this research is to automate the process of discovering, neutralizing and repairing software bugs and vulnerabilities. As part of this goal, we build components of a continuous automatic improvement system that can automatically search for errors and generate patches that repair the encountered errors. By removing the human from the loop, patch generation time can be reduced, patch robustness improved, leading to fewer unpatched systems. The systems that we developed during this program lay the foundation for future automatic program repair systems that can significantly reducing the time and effort required to deal with software defects.

15. SUBJECT TERMS
software protection, automatic program repair, software bug finding, automatic input rectification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Philip Mumford
Unclassified	Unclassified	Unclassified	SAR	118	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Table of Contents

List of Figures

List of Tables

1 Summary

Despite decades of effort, defect triage and correction remains a central concern in software engineering. Indeed, modern software projects contain so many defects, and the cost of correcting defects remains so large, that projects typically ship with a long list of known but uncorrected defects. Consequences of this unfortunate situation include pervasive security vulnerabilities and the diversion of resources that would be better devoted to other, more productive, activities.

The goal of this research is to automate the process of discovering, neutralizing and repairing software bugs and vulnerabilities. As part of this goal, we build components of a continuous automatic improvement system that can automatically search for errors and generate patches that repair the encountered errors. By removing the human from the loop, patch generation time can be reduced, patch robustness improved, leading to fewer unpatched systems.

The systems that we developed during this program lay the foundation for future automatic program repair systems that can significantly reducing the time and effort required to deal with software defects.

2 Methods, Assumptions and Procedures

For all the research we performed in this program, we adopted an experimental approach. We chose to evaluate our developed systems on realistic real-world applications and formats to better understand their direct applicability to complex systems. All of the systems that we developed run on open source infrastructure (e.g., Linux) and do not require proprietary software to build and run. We made part of our software available for download via the Internet.

In general, we observed the results we obtained and the general process of obtaining these results and used them to drive further development. We also identified any weaknesses or missing pieces and worked towards remedying the weaknesses and filling in any missing pieces.

During the course of the project we devoted a major effort to integrating and evaluating the various different components. Our integration efforts focused on developing software to connect the different components. Once the software was developed we tested it and updated it as the tests indicated was necessary. We evaluated our techniques by applying them to different exploits. During this process we observed any deficiencies and developed techniques that addressed these deficiencies.

3 Introduction

Software errors and vulnerabilities in server applications are a significant problem for preserving system integrity and availability. The accepted wisdom is to use a multitude of tools, such as diligent software development strategies, dynamic bug finders and static analysis tools in an attempt to eliminate as many bugs as possible.

However, experience has shown that it is very hard to achieve bug-free software. As a result, even under the best of circumstances, buggy software is deployed and developers face a constant and time-consuming battle of creating and releasing patches fast enough to fix newly discovered bugs. Patches can take days if not weeks to create, and it is not uncommon for systems to continue running unpatched applications long after an exploit of a bug has become well-known.

The goal of this research is to automate the process of discovering, neutralizing and repairing software bugs and vulnerabilities. In other words, to build a continuous automatic improvement systems that can automatically search for errors and generate patches that repair the encountered errors. By removing the human from the loop, patch generation time can be reduced, patch robustness improved, leading to fewer unpatched systems.

Our approach towards building continuous automatic improvement systems, revolves around three core thrusts as shown in Figure 1:

- **Vulnerability Discovery**
- **Vulnerability Isolation and Neutralization**
- **Vulnerability Repair**

3.1 Vulnerability Discovery

Previous techniques such as Fuzzing [15, 17] and concolic execution [46, 26, 36, 27] have been shown to be effective in discovering errors in the initial input parsing stages of computations, but have had little to no success in exposing errors that lie deep within the program.

As part of our research in automating vulnerability discovery we have researched and developed, under the CIDER project, a new technique and system, DIODE

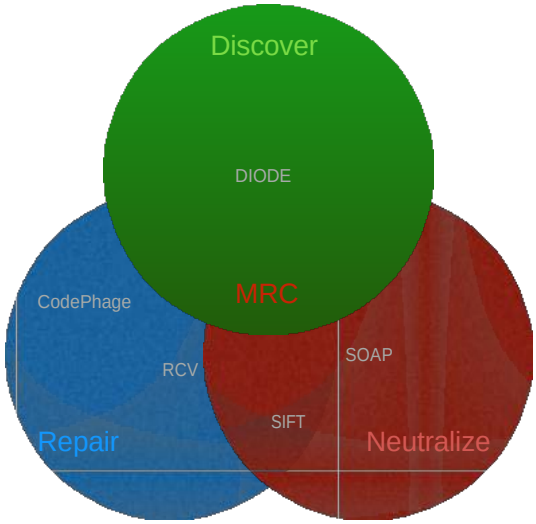


Figure 1: Overview of CIDER's Approach to Self-Healing Systems

(Directed Integer Overflow Discovery Engine) [52], for automatically generating inputs that trigger integer overflow errors at critical sites. DIODE starts with a target site (such as a memory allocation site) and a target value (such as the size of the allocated memory block). It then uses symbolic execution to obtain an target expression that characterizes how the program computes the target value as a function of the input. It then transforms the target expression to obtain a target constraint. If the input 1) satisfies the target constraint while 2) causing the program to execute the target site, then it will trigger the error.

DIODE shows that discovering and targeting specific potentially vulnerable program sites can effectively expose such deep errors. One of the keys to success is new techniques that work appropriately with sanity and blocking checks to obtain inputs that can successfully traverse these obstacles to reach the target site. The success of DIODE in exposing integer overflow vulnerabilities opens up the field to the further development of other targeted techniques that work effectively with sanity and blocking checks to expose deep errors.

DIODE works with off-the-shelf, production x86 binaries. Our results show that, for our benchmark set of applications, and for every target memory alloca-

tion site exercised by our seed inputs (which the applications process correctly with no overflows), either 1) DIODE is able to generate an input that triggers an overflow at that site or 2) there is no input that would trigger an overflow for the observed target expression at that site.

3.2 Vulnerability Isolation and Neutralization

Errors and security vulnerabilities in software often occur in infrequently executed program paths triggered by atypical inputs. A standard way to ameliorate this problem is to use an anomaly detector that filters out such atypical inputs. The goal is to ensure that the program is only presented with standard inputs that it is highly likely to process without errors. A drawback of this technique is that it can filter out desirable, benign, but atypical inputs along with the atypical malicious inputs, thereby denying the user access to useful inputs.

We propose a new technique, *automatic input rectification*. Instead of rejecting atypical inputs, the input rectifier modifies the input so that it is typical, then presents the input to the application, which then processes the input. We have three goals: a) present typical inputs (which the application is highly likely to process correctly) to the application unchanged, b) render any malicious inputs harmless by eliminating any atypical input features that may trigger errors or security vulnerabilities, while c) preserving most, if not all, of the desirable behavior for benign atypical inputs. A key empirical observation that motivates our technique is the following:

Production software is usually tested on a large number of inputs. Standard testing processes ensure that the software performs acceptably on such inputs. We refer to such inputs as *typical inputs* and the space of such typical inputs as the *comfort zone* [51] of the application. On the other hand, inputs designed to exploit security vulnerabilities (i.e., *malicious inputs*) often lie outside the comfort zone. If the rectifier is able to automatically detect inputs that lie outside the comfort zone and map these inputs to corresponding meaningfully close inputs within the comfort zone, then it is possible to a) prevent attackers from exploiting the vulnerability in the software while b) preserving the ability of the user to access desirable data in atypical inputs (either benign or malicious).

We present two systems for implementing automatic input rectification: SOAP [41] and SIFT [43].

SOAP (Sanitization Of Anomalous inPuts), is an automatic input rectification system designed to prevent *overflow vulnerabilities* and other memory addressing errors. SOAP first learns a set of constraints over typical inputs that characterize a comfort zone for the application that processes those inputs. It then takes the constraints and automatically generates a rectifier that, when provided with an input, automatically produces another input that satisfies the constraints. Inputs that already satisfy the constraints are passed through unchanged; inputs that do not satisfy the constraints are modified so that they do.

SOAP is a reactive system that has some drawbacks such as incomplete coverage and does not protect the application until it is attacked. SIFT is a proactive system, SIFT, for generating filters that discard inputs that may cause integer overflow errors at memory allocation and block copy sites. Unlike previous reactive systems, SIFT proactively analyzes the program before it executes to generate filters that take all execution paths into consideration. SIFT can therefore nullify exploits that target unknown vulnerabilities (i.e., zero-day attacks).

The combination of SOAP and SIFT provides support for applying automatic input rectification for systems statically (where access to source code is available) and dynamically.

3.3 Repair

Despite decades of effort, defect triage and correction remains a central concern in software engineering. Indeed, modern software projects contain so many defects, and the cost of correcting defects remains so large, that projects typically ship with a long list of known but uncorrected defects. Consequences of this unfortunate situation include pervasive security vulnerabilities and the diversion of resources that would be better devoted to other, more productive, activities. Automatic program repair holds out the promise of significantly reducing the time and effort required to deal with software defects.

Under the CIDER project, we developed three automatic repair systems: CodePhage [59], RCV [44], and SPR [42].

Code Phage (CP), a system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Because CP works with binary donors with no need for source code or symbolic information, it supports a wide range

of use cases. To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.

RCV, for enabling software applications to survive divide-by-zero and null-dereference errors. RCV operates directly on off-the-shelf, production, stripped x86 binary executables. RCV implements recovery shepherding, which attaches to the application process when an error occurs, repairs the execution, tracks the repair effects as the execution continues, contains the repair effects within the application process, and detaches from the process after all repair effects are flushed from the process state. RCV therefore incurs negligible overhead during the normal execution of the application.

SPR, a new program repair system that uses a novel staged program repair strategy to efficiently search a rich search space of candidate repairs. Three key techniques work synergistically together to enable SPR to generate successful repairs for a range of software defects. Together, these techniques enable SPR to generate correct repairs for over five times as many defects as previous systems evaluated on the same benchmark sets

4 Automatic Input Rectification

Errors and security vulnerabilities in software often occur in infrequently executed program paths triggered by atypical inputs. A standard way to ameliorate this problem is to use an anomaly detector that filters out such atypical inputs. The goal is to ensure that the program is only presented with standard inputs that it is highly likely to process without errors. A drawback of this technique is that it can filter out desirable, benign, but atypical inputs along with the atypical malicious inputs, thereby denying the user access to useful inputs.

4.1 Input Rectification

(a) The original image

(b) The rectified image

Figure 2: An example image truncated by the rectification.

We propose a new technique, *automatic input rectification*. Instead of rejecting atypical inputs, the input rectifier modifies the input so that it is typical, then presents the input to the application, which then processes the input. We have three goals: a) present typical inputs (which the application is highly likely to process correctly) to the application unchanged, b) render any malicious inputs harmless by eliminating any atypical input features that may trigger errors or security vulnerabilities, while c) preserving most, if not all, of the desirable behavior for benign atypical inputs. A key empirical observation that motivates our technique is the following:

Production software is usually tested on a large number of inputs. Standard testing processes ensure that the software performs acceptably on such inputs. We refer to such inputs as *typical inputs* and the space of such typical inputs as the *comfort zone* [51] of the application. On the other hand, inputs designed to exploit security vulnerabilities (i.e., *malicious inputs*) often lie outside the comfort zone. If the rectifier is able to automatically detect inputs that lie outside the comfort zone and map these inputs to corresponding meaningfully close inputs within the comfort zone, then it is possible to a) prevent attackers from exploiting the vulnerability in the software while b) preserving the ability of the user to access desirable data in atypical inputs (either benign or malicious).

We present SOAP (Sanitization Of Anomalous inPuts), an automatic input rectification system designed to prevent *overflow vulnerabilities* and other memory addressing errors. SOAP first learns a set of constraints over typical inputs that characterize a comfort zone for the application that processes those inputs. It then takes the constraints and automatically generates a rectifier that, when provided with an input, automatically produces another input that satisfies the constraints. Inputs that already satisfy the constraints are passed through unchanged; inputs that do not satisfy the constraints are modified so that they do.

4.2 Potential Advantages of Automatic Input Rectification

Input rectification has several potential advantages over simply rejecting malicious or atypical inputs that lie outside the comfort zone:

- **Desirable Data in Atypical Benign Inputs:** Anomaly detectors filter out atypical inputs even if they are benign. The result is that the user is completely denied access to data in atypical inputs. Rectification, on the

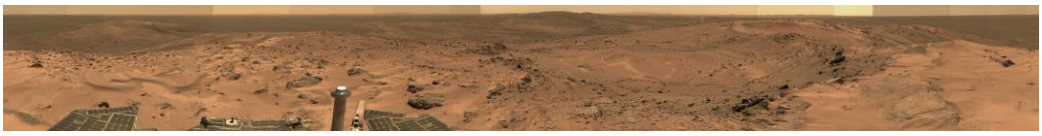
other hand, passes the rectified input to the application for presentation to the user. Rectification may therefore deliver much or even all of the desirable data present in the original atypical input to the user.

- **Desirable Data in Malicious Inputs:** Even a malicious input may contain data that is desirable to the user. Common examples include videos and web pages with embedded malicious content. Rectification may eliminate the exploits while preserving much of the desirable input from the original input. In this case the rectifier enables the user to safely access the desirable data in the malicious input.
- **Error Nullification:** Even if they are not malicious, atypical inputs may expose errors that prevent the application from processing them successfully. In this case rectification may nullify the errors so that the application can deliver most if not all of the desirable data in the input to the user.

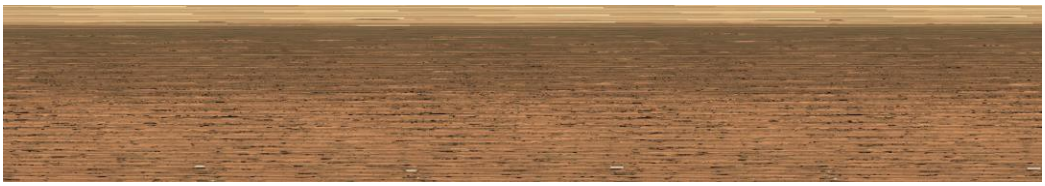
4.3 The Input Rectification Technique

SOAP operates on the parse tree of an input, which divides the input into a collection of (potentially nested) fields. Each field may contain an integer value, a string, or unparsed raw data bytes. SOAP infers and enforces 1) upper bound constraints on the values of integer fields, 2) constraints that capture whether or not an integer field must be non-negative, 3) upper bound constraints on the lengths of string or raw data byte fields, and 4) field length indicator constraints that capture relationships between the values of integer fields and the lengths of string or raw data fields.

The dynamic taint analysis [28, 49, 35] engine of SOAP first identifies input fields that are related to critical operations during the execution of the application such as memory allocations and memory writes. The learning engine of SOAP then automatically infers constraints on these fields based on a set of training inputs. When presented with an atypical input that violates these constraints, the rectifier of SOAP automatically modifies input fields iteratively until all of the constraints are satisfied.



(a) The original image



(b) The rectified image

Figure 3: An example image twisted by the rectification

4.4 Nested Fields in Input Files

One of the key challenges in input rectification is the need to deal with nested fields. In general, input formats are in tree structures containing arbitrarily nested fields. Inferring correlated constraints is hard, because our algorithm must consider relationships of multiple fields at different levels in the tree.

Nested input fields also complicate the rectification. Changing one field may cause the file to violate constraints associated with enclosing fields. To produce a consistent rectified input, the rectifier must therefore apply a cascading sequence of modifications to correlated constraints as its constraint enforcement actions propagate up or down the tree of nested fields.

4.5 Key Questions

We identify several key questions that are critical to the success of the input rectification technique:

- **Learning:** Is it possible to automatically learn an effective set of constraints from a set of typical non-malicious or benign inputs?
- **Rectification Percentage:** Given a set of learned constraints, what percentage of previously unseen benign inputs fail to satisfy the constraints and will therefore be modified by the rectifier?

- **Rectification Quality:** What is the overall quality of the outputs that the application produces when given benign inputs that the rectifier has modified to conform to the constraints?
- **Security:** Does the rectifier effectively protect the application against inputs that exploit errors and security vulnerabilities?

We investigate these questions by applying SOAP to rectify inputs for five large software applications. The input formats of these applications include three image types (PNG, TIFF, JPEG), wave sound (WAV) and Shockwave flash video (SWF). We evaluate the effectiveness of our rectifier by performing the following experiments:

- **Input Acquisition:** For each application, we acquire a set of inputs from the Internet.
- **Benign Input Acquisition:** We run each application on each input in its set and filter out any inputs that cause the application to crash. The resulting set of inputs is the *benign inputs*. Because all of our applications are able to process all of the inputs without errors, the set of benign inputs is the same as the original set.
- **Training and Test Inputs:** We next randomly divide the inputs into two sets: the *training set* and the *test set*.
- **Potentially Malicious Inputs:** We search the CVE security database [2] and previous security papers to obtain malicious inputs designed to trigger errors in the applications.
- **Learning:** We use the training set to automatically learn the set of constraints that characterize the comfort zone of the application.
- **Atypical Benign Inputs:** For each application, we next compute the percentage of the benign inputs that violate at least one of the learned constraints. We call such inputs *atypical benign inputs*. For our set of applications, the percentage of atypical benign inputs ranges from 0% to 1.57%.
- **Quality of Rectified Atypical Inputs:** We evaluate the quality of the rectified atypical inputs by paying people on Amazon Mechanical Turk [1] to evaluate their perception of the difference between 1) the output that the application produces when given the original input and 2) the output that the application produces when given the rectified version of the original input. Specifically, we paid people to rank the difference on a scale from 0 to 3, with 0 indicating completely different outputs and 3 indicating no perceived difference. The average scores for over 75% of the atypical inputs are greater than 2.5, indicating that Mechanical Turk workers perceive the

outputs for the original and rectified inputs to be very close.

- **Security Evaluation:** We verified that the rectified versions of malicious inputs for each of these applications were processed correctly by the application.
- **Manual Code Analysis:** For each of the malicious inputs, we manually identify the root cause of the vulnerability that the malicious input exploited. We then examined the set of learned constraints and verified that if an input satisfies the constraints, then it will not be able to exploit the vulnerabilities.

4.6 Understanding Rectification Effects

We examined the original and rectified images or videos for all test input files that the rectifier modified. All of these files are available at:

https://sites.google.com/site/inputrectification/home

For the majority of rectified inputs (83 out of 110 inputs), the original and rectified images or videos appear identical. The average Mechanical Turk rating for such images or videos was between 2.5 and 3.0. We attribute this phenomenon to the fact that the rectifier often modifies fields (such as the name of the author of the file) that are not relevant to the core functionality of the application and therefore do not visibly change the image or video presented to the user. The application must nevertheless parse and process these fields to obtain the desirable data in the input file. Furthermore, since these fields are often viewed as tangential to the primary purpose of the application, the code that parses them may be less extensively tested and therefore more likely to contain errors.

Figure 2, 3 and 4 show examples of image files that are visibly changed by rectification. For some of the rectified image inputs (8 of 53 image inputs), the rectifier truncates part of the image, leaving a strip along the bottom of the picture (see Figure 2). For the remaining inputs (19 of 110), the rectifier changes fields that control various aspects of core application functionality, for example, the alignment between pixels and the image size (see Figure 3), the color of the image (see Figure 4), or interactive aspects of videos. The average Mechanical Turk rating for such images or videos varied depending on the severity of the effect. In all cases the application was able to successfully process the rectified inputs without error to present the remaining data to the user.

(a) The original image

(b) The rectified image

Figure 4: An example image whose color is changed by the rectification.

4.7 Contributions

We make the following contributions:

- **Basic Concept:** We propose a novel technique for dealing with anomalous and potentially malicious inputs, namely, automatic input rectification, and an prototype implementation, SOAP, which demonstrates the effectiveness of the technique.
- **Constraint Inference:** We show how to use dynamic taint analysis and a constraint inference algorithm to automatically infer safety constraints. This constraint inference algorithm operates correctly to infer correlated constraints for hierarchically structured input files with nested fields.
- **Rectification Algorithm:** We present an input rectification algorithm that systematically enforces safety constraints on inputs while preserving as much of the benign part of the input as possible. Because it is capable of enforcing correlated constraints associated with nested input fields, this algorithm is capable of rectifying hierarchically structured input files.

4.8 Motivating Example

Figure 5 presents source code from Dillo 2.1, a lightweight open source web browser. Dillo uses libpng to process PNG files. The libpng callback function *Png_datainfo_callback()* shown in Figure 5 is called when Dillo starts to load a PNG file. The function contains an integer overflow bug at line 20, where the multiplication calculates the size of the image buffer allocated for future callbacks. Because *png→rowbytes* is proportional to the image width, arithmetic integer overflow will occur when opening a PNG image with maliciously large width and height values. This error causes Dillo to allocate a significantly smaller buffer than required.

Dillo developers are well aware of the potential for overflow errors. In fact, the code contains a check of the image size at lines 10-11 to block large images. Unfortunately, their bound check has a similar integer overflow problem. Specific large width and height values can also cause an overflow at line 10, and thus bypass the check. To nullify the above Dillo error, SOAP performs following steps:

- **Understand Input Format:** SOAP first parses a PNG image file into a

```
1  //Dillo's libpng callback
2  static void
3  Png_datainfo_callback(png_structp png_ptr, ...)
4  {
5    DilloPng *png;
6    ...
7    png = png_get_progressive_ptr(png_ptr);
8    ...
9    /* check max image size */
10   if (abs(png→width*png→height) >
11     IMAGE_MAX_W * IMAGE_MAX_H) {
12       ...
13       Png_error_handling(png_ptr, "Aborting...");
14       ...
15   }
16   ...
17   png→rowbytes = png_get_rowbytes(png_ptr, info_ptr);
18   ...
19   png→image_data = (uchar_t *) dMalloc(
20     png→rowbytes * png→height);
21   ...
22 }
```

Figure 5: The code snippet of Dillo libpng callback (png.c). Highlighted code is the root cause of the overflow bug.

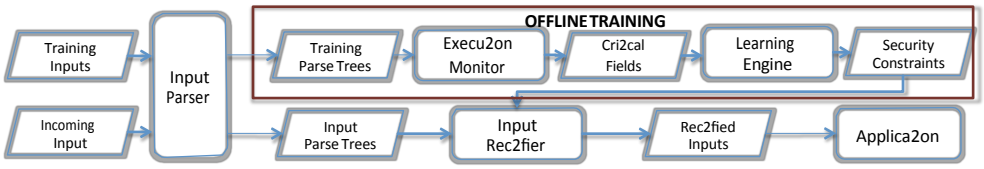


Figure 6: The architecture of automatic input rectification system.

collection of input fields (as shown in Figure 7), so that SOAP knows which input bytes in the PNG image file correspond to the image width and height in the above example.

- **Identify Critical Fields:** SOAP monitors the execution of Dillo to determine that values in the image width and height fields flow into the variables *png→width* and *png→height*. These two variables influence a memory allocation statement at lines 19-20. Thus SOAP marks width and height in PNG images as critical fields, which can potentially cause dangerous overflow.
- **Infer Constraints:** SOAP next infers constraints over the critical fields. Specifically, SOAP processes the benign training PNG images to use the maximum image width and height values that appear in these inputs as their upper bounds. Figure 8 presents more examples of constraints for PNG images.
- **Rectify Atypical Inputs:** When it encounters an atypical input whose width or length fields are larger than the inferred bound, SOAP enforces the bound by changing the field to the inferred bound. Note that such changes may, in turn, cause other constraints (such as the length of another field involved in a correlated relation with the modified field) to be violated. SOAP therefore rectifies violated constraints until all constraints are satisfied.

Both critical field identification and constraint inference are done offline. Once SOAP generates safety constraints for the PNG format, it can automatically rectify new incoming PNG images.

4.9 Design

SOAP has four components: the *input parser*, the *execution monitor*, the *learning engine*, and the *input rectifier*. The components work together cooperatively to enable automatic input rectification (see Figure 6). The execution monitor and the learning engine together generate safety constraints offline, before the input rectifier is deployed:

- **Input parser:** The input parser ***understands input formats***. It transforms raw input files into syntactic parse trees for the remaining components to process.

- **Execution Monitor:** The execution monitor uses taint tracing to analyze the execution traces of an application. It ***identifies critical input fields*** that influence sensitive operations including memory allocations and memory writes.
- **Learning Engine:** The learning engine starts with a set of benign training inputs. It ***infers safety constraints*** based on the values of the fields in these training inputs. Safety constraints define the comfort zone of the application.
- **Input Rectifier:** The input rectifier ***rectifies atypical inputs*** to enforce safety constraints. The rectification algorithm modifies the input iteratively until it satisfies all constraints.

4.9.1 Input Parser

As shown in Figure 6, the input parser transforms an arbitrary input into a general syntactic parse tree that can be easily consumed by the remaining components. In the syntactic parse tree, only leaf fields are directly associated with input data. Each leaf field has a type, which can be integer, string or raw bytes, while each non-leaf field contains several child fields which together forms a coarser semantic chunk. The parse tree also contains low-level specification information, for example, how the input file encodes these values. The input rectifier uses this information when modifying input fields.

Figure 7 presents an example of a leaf field inside a parse tree for a PNG image file. The leaf field identifies the location of the data in the input file. It also contains a descriptor that specifies various aspects of the field, such as the value stored in the field, the name of the field, and the encoding information such as whether the value is stored in big endian or little endian form. The input rectifier uses this information in the descriptor when modifying the field.

As shown in Figure 7, the field name is similar to the path name in a file system, which corresponds the position of the field inside the tree. Each field also stores additional information to help the rectifier modify the input, including the endianness, the encoding method and the offset position of corresponding bytes in the input.

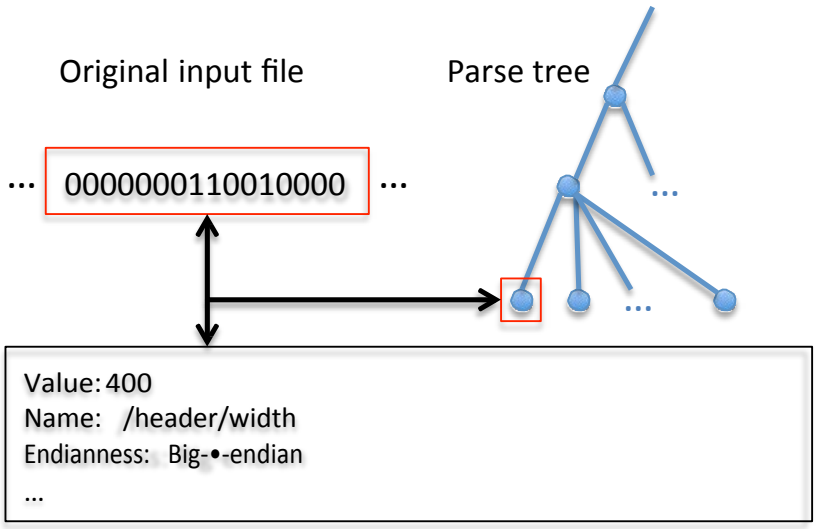


Figure 7: An example of syntax parse tree.

4.9.2 Execution Monitor

The execution monitor is responsible for identifying the critical input fields that are involved in the learned constraints. Because large data fields may trigger memory buffer overflows, the execution monitor treats all variable-length data fields as critical. Integer fields present a more complicated scenario. Integer fields that influence the addresses of memory writes or the values used at memory allocation sites (e.g., calls to *malloc*() and *calloc*()) are relevant for our target set of errors. Other integer fields (for example, control bits or checksums) may not affect relevant program actions.

The SOAP execution monitor uses dynamic taint analysis [28, 49] to compute the set of critical integer fields. Specifically, SOAP considers an integer field to be critical if the dynamic taint analysis indicates that the value of the field may influence the address of memory writes or values used at memory allocation sites. The execution monitor uses an automated greedy algorithm to select a subset of the training inputs for the runs that determine the critical integer fields. The goal is to select a small set of inputs that 1) minimize the execution time required to find the integer fields and 2) together cover all of the integer fields that may appear in the input files.

```
1  /header/width <= 1920
2  /header/width >= 0
3  sizebits(/text/text) <= 21112
4  /text/size * 8 == sizebits(/text/keyword)
5      + sizebits(/text/text)
```

Figure 8: A subset of constraints generated by SOAP for PNG image files.

The execution monitor currently tracks data dependences only. This approach works well for our set of applications, eliminating 58.3%-88.7% of integer fields from consideration. It would be possible to use a taint system that tracks control dependences [24] as well.

4.9.3 Learning Engine

The learning engine works with the parse trees of the training inputs and the specification of critical fields as identified by the execution monitor. It uses this information to infer safety constraints over critical fields (see offline training box in Figure 6).

Safety Constraints: Overflow bugs are typically exploited by large data fields, extreme values, negative entries or inconsistencies of multiple fields. SOAP infers both bound constraints and length indicator constraints. Bound constraints are associated with individual fields, which bound values of critical integer fields and sizes of data fields in incoming inputs. Length indicator constraints (i.e., an integer field that indicates the actual length of a data field) are correlated constraints associated with multiple fields.

Figure 8 presents several examples of constraints that SOAP infers for PNG image files. Specifically, SOAP infers upper bounds of integer fields (line 1), non-negativity of integer fields (line 2), upper bounds of lengths of data fields (line 3), and length indicator constraints between values and lengths of parse tree fields (lines 4-5 in Figure 8).

These constraints enable the rectification system to eliminate extreme values in integer fields, overly long data fields, and inconsistencies between the specified and actual lengths of data fields in the input. When properly inferred and enforced, these constraints enable the rectifier to nullify our target vulnerabilities in the protected programs.

Note that once SOAP infers a set of safety constraints for one input format, it

may use these constraints to rectify inputs for any application that reads inputs in that format. This is useful when multiple different applications are vulnerable to the same exploit. For example, both Picasa [16] and ImageMagick [9] are vulnerable to the same integer overflow exploit (see Section 4.11). A single set of inferred constraints enables SOAP to nullify the vulnerability for both applications.

Inferring Bound Constraints: SOAP infers three kinds of bound constraints: upper bounds of lengths of data fields, upper bounds of integer fields, and whether integer fields are non-negative. SOAP sets the maximum length of a data field that appeared in training inputs as the upper bound of its length. SOAP sets the maximum value of an integer field in training inputs as the upper bound of its value. SOAP also sets an integer field to be non-negative if it is never negative in all training inputs. SOAP infers all these constraints with a single traversal of the parse tree of each training input.

Inferring Length Indicators: Inferring length indicator constraints is challenging, because of the presence of nested fields in hierarchical input format. For example, an integer field may indicate the total length of several big fields which recursively enclose many sub-fields. Moreover, such constraints may appear at various levels in the input parse tree.

SOAP infers a length indicator field f which is associated with the total length of consecutive children of the parent field of f. For instance, lines 4-5 in Figure 8 present a length indicator constraint. The constraint states that the value of "`/text/size`" is the total length of "`/text/keyword`" and "`/text/text`", which are two consecutive children of "`/text`".

SOAP constraint learning algorithm first enumerates all possible field combinations for length indicator constraints, and initially assumes that all of these constraints are true. When processing each training input, the algorithm eliminates constraints that do not hold in the input. Our algorithm can be extended to infer other kinds of correlated constraints. More details and pseudo-code of our learning algorithm can be found in our technical report [41].

4.9.4 Input Rectifier

Given safety constraints generated by the learning engine and a new input, the input rectifier rectifies the input if it violates safety constraints (see Figure 6).

The main challenge in designing the input rectifier is enforcing safety constraints while preserving as much desirable data as possible.

Our algorithm is designed around two principles: 1) It enforces constraints only by modifying integer fields or truncating data fields—it does not change the parse tree structure of the input. 2) At each step, it finds a single violated constraint and applies a minimum modification or truncation to satisfy the violated constraint.

Nested input fields further complicate rectification, because changing one field may cause the file to violate correlated constraints associated with enclosing or enclosed fields at other levels. Thus our algorithm must iteratively continue the rectification process until there are no more violated constraints. In our experiment, SOAP enforces as many as five correlated constraints on some rectified input files.

Our algorithm has a main loop that iteratively checks the input against learned constraints. The main loop exits when the input no longer violates any safety constraints. At each iteration, it applies various rectification actions depending on the violated constraints:

- **Upper bounds of integer fields:** Our algorithm changes the value of an integer field to the learned upper bound, if the input violates the upper bound constraint of the field.
- **Non-negativities of integer fields:** Our algorithm changes the value of an integer field to 0, if the input violates the non-negative constraint of the field.
- **Length upper bounds of data fields:** Our algorithm truncates a data field to its length upper bound, if the input violates the length upper bound constraint of the data field.
- **Length indicator constraints:** Our algorithm changes the value of the length indicator field to the actual length of the data field, if the value is greater than the actual length. Our algorithm truncates the data fields to the length indicated by the corresponding integer field, if the data is longer than the indicated length. Note that the length indicator constraints may be violated due to previous fixes for other constraints. Our algorithm cannot increase the value of the length indicator field or increase the length of the data field here, which will roll back previous fixes.

Note that, because the absolute values of integer fields and the lengths of data

fields always decrease at each iteration, this algorithm will always terminate. Note also that, because the algorithm truncates a minimum amount of data each iteration, the algorithm attempts to minimize the total amount of discarded data. More details and pseudo-code of the rectification algorithm can be found in our technical report [41].

Checksum: SOAP appropriately updates checksums after the rectification. SOAP currently relies on the input parser to identify the fields that store checksums and the method used to compute checksums. After the rectification algorithm terminates, SOAP calculates the new checksums and appropriately updates checksum fields. It is also possible to use an more automatic checksum repair technique [56].

4.10 Implementation

The SOAP learning engine and input rectifier are implemented in Python. The execution monitor is implemented in C based on Valgrind [47], a dynamic binary instrumentation framework. The input parser is implemented with Hachoir [8], a manually maintained Python library for parsing binary streams in various formats. SOAP is able to process any file format that Hachoir supports. Because SOAP implements an extensible framework, it can work with additional parser components that allow to support other input formats.

4.11 Quantitative Evaluation

We next present a quantitative evaluation of SOAP using five popular media applications. Specifically, the following questions drive our evaluation:

1. Is SOAP effective in nullifying errors?
2. How much desirable data does rectification preserve?
3. How does the amount of training inputs affect SOAP's ability to preserve desirable data?

Applications and Errors: We use SOAP to rectify inputs for five applications: Swfdec 0.5.5 (a shockwave player) [18], Dillo 2.1 (a lightweight browser) [4],

Application	Sources	Fault	Format	Position	Related constraints
Swfdec	Buzzfuzz	X11 crash	SWF	XCreatePixMap	/rect/xmax ≤ 57600 /rect/ymax ≤ 51000
Swfdec	Buzzfuzz	overflow/crash	SWF	jpeg.c:192	/sub jpeg/.../width ≤ 6020 /sub jpeg/.../height ≤ 2351
Dillo	CVE-2009-2294	overflow/crash	PNG	png.c:142 png.c:203	/header/width ≤ 1920 /header/height ≤ 1080
ImageMagick	CVE-2009-1882	overflow/crash	JPEG,TIFF	xwindow.c:5619	/ifd[..]/img_width/value ≤ 14764 /ifd[..]/img_height/value ≤ 24576
Picasa	TaintScope	overflow/crash	JPEG,TIFF	N/A	/start_frame/content/width ≤ 15941 /start_frame/content/height ≤ 29803
VLC	CVE-2008-2430	overflow/crash	WAV	wav.c:147	/format/size ≤ 150

Figure 9: SOAP result summary

	Rectification Statistics						Running Time			
Inp.	App.	Train	Test	Field (Distinct)	Rectified	Avg. P_{loss}	Mean	Parse	Rect.	Per field
SWF	Swfdec	3620	3620	5550.2 (98.17)	57 (1.57%)	N/A	531ms	443ms	88ms	0.096ms
PNG	Dillo	1496	1497	306.8 (32.3)	0 (0%)	0%	23ms	19ms	4ms	0.075ms
JPEG	IMK, Picasa	3025	3024	298.2 (75.5)	42 (1.39%)	0.08%	24ms	21ms	3ms	0.080ms
TIFF	IMK, Picasa	870	872	333.5 (84.5)	11 (1.26%)	0.50%	31ms	26ms	5ms	0.093ms
WAV	VLC	5488	5488	17.1 (16.8)	11 (0.20%)	0%	1.5ms	1.3ms	0.2ms	0.088ms

Figure 10: Benchmarks and numerical results of SOAP experiment

ImageMagick 6.5.2-8 (an image processing toolbox) [9], Google Picasa 3.5 (a photo managing application) [16], and VLC 0.8.6h (a media player) [21].

Figure 9 presents a description of each error in each application. In sum, all of these applications consume inputs that (if specifically crafted) may cause the applications to incorrectly allocate memory or perform an invalid memory access. The input file formats for these errors are the SWF Shockwave Flash format; the PNG, JPEG, and TIF image formats; and the WAV sound format.

Malicious inputs: We obtained six input files from CVE database [2], Buzzfuzz project [35] and TaintScope project [56]. Each input targets a distinct error (see Figure 9) in at least one of these applications.

Benign inputs: We implemented a web crawler to collect input files for each format (see Figure 10 for the number of collected inputs for each input format). Our web crawler uses Google's search interface to acquire a list of pages that contain at least one link to a file of a specified format (e.g., SWF, JPEG, or WAV). The crawler then downloads each file linked within each page. We verified that all of these inputs are benign, i.e., the corresponding applications successfully processed these inputs. For each format, we randomly partitioned these inputs into

two sets, the training set and the test set (see Figure 10).

4.11.1 Nullifying Vulnerabilities

We next evaluate the effectiveness of SOAP in nullifying six vulnerabilities in the benchmark applications (see Figure 9). We first applied the trained SOAP rectifier to the obtained malicious inputs. The rectifier detected that all of these inputs violated at least one safety constraint. It rectified all violated constraints to produce six corresponding rectified inputs. We verified that the applications processed the rectified inputs without error and none of the rectified inputs exploited the vulnerabilities. We next discuss the interactions between the inputs and the root cause of each vulnerability.

Flash video: The root cause of the X11 crash error in Swfdec is a failure to check for large Swfdec window sizes as specified in the input file. If this window size is very large, the X11 library will allocate an extremely large buffer for the window and Swfdec will eventually crash. SOAP nullifies this error by enforcing the constraints that */rect/xmax* $\leq$ *57600* and */rect/ymax* $\leq$ *51000*, which limit the window to a size that Swfdec can handle. In this way, SOAP ensures that no rectified input will be able to exploit this error in Swfdec.

The integer overflow bug in Swfdec occurs when Swfdec calculates the required size of the memory buffer for JPEG images embedded within the SWF file. If the SWF input file contains a JPEG image with abnormally large specified width and height values, this calculation will overflow and Swfdec will allocate a buffer significantly smaller than the required size. When SOAP enforces the learned safety constraints, it nullifies the error by limiting the size of the embedded image. No rectified input will be able to exploit this error.

Image: Errors in Dillo, ImageMagick and Picasa have similar root causes. A large PNG image with crafted width and height can exploit the integer overflow vulnerability in Dillo (see Section 4.8). The same malicious JPEG and TIFF images can exploit vulnerabilities in both ImageMagick and Picasa Photo Viewer. ImageMagick does not check the size of images when allocating an image buffer for display at *magick/xwindow.c:5619* in function *XMakeImage()*. Picasa Photo Viewer also mishandles large image files [56]. By enforcing the safety constraints, SOAP limits the size of input images and nullifies these vulnerabilities.

Sound: VLC has an overflow vulnerability when processing the format chunk of a WAV file. The integer field */format/size* specifies the size of the format chunk (which is less than 150 in typical WAV files). VLC allocates a memory buffer to hold the format chunk with the size of the buffer equal to the value of the field */format/size* plus two. A malicious input with a large value (such as 0xfffffffe) in this field can exploit the overflow vulnerability. By enforcing the constraint */format/size* $\leq$ *150*, SOAP limits the size of the format chunk in WAV file and nullifies this vulnerability.

These results indicate that SOAP effectively nullifies all six vulnerabilities. Our inspection of the source code indicates that the inferred safety constraints nullify the root causes of all of the vulnerabilities so that no input, after rectification, can exploit the vulnerabilities.

4.11.2 Data Loss

We next compute a quantitative measure of the effect of rectification on data loss. For each input format, we first apply the SOAP rectifier to the test inputs. We report the average data loss percentage of all test inputs for each format. We use the following formula to compute the data loss percentage of each rectified input:

$$P_{loss} = \frac{D_{loss_i}}{D_{tot_i}}$$

D_{tot_i} measures the amount of desirable data before rectification and D_{loss_i} measures the amount of desirable data lost in the rectification process. For JPEG, TIFF and PNG files, D_{tot_i} is the number of pixels in the image and D_{loss_i} is the number of changed pixels after rectification. For WAV files, D_{tot_i} is the number of frames in the sound file and D_{loss_i} is the number of changed frames after rectification. Because SWF files typically contain interactive content such as animations and dynamic objects that respond to user inputs, we did not attempt to develop a corresponding metric for these files.

Result Interpretation: Figure 10 presents rectification results of the test inputs of each input format. First, note that the vast majority of the test inputs satisfy all of the learned constraints and are therefore left unchanged by the rectifier. Note also that both PNG and WAV have zero desirable data loss — PNG because the

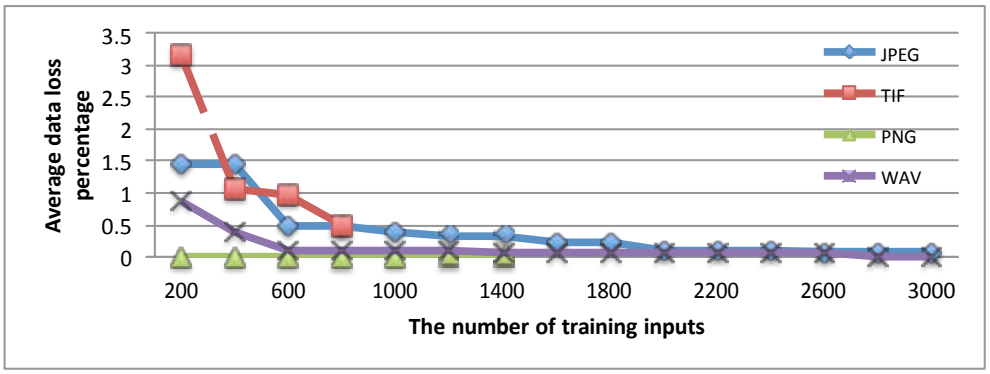


Figure 11: Average data loss percentage curves under different sizes of training

rectifier did not modify any test inputs, WAV because the modifications did not affect the desirable data. For JPEG and TIFF, the average desirable data loss is less than 0.5%.

One of the reasons that the desirable data loss numbers are so small is that rectifications often change fields (such as the name of the author of the data file or the software package that created the data file) that do not affect the output presented to the user. The application must nevertheless parse and process these fields to obtain the desirable data in the input file.

4.11.3 Size of Training Input Set

We next investigate how the size of the training input set affects the effectiveness of the rectification. Intuitively, we expect that using less training inputs will produce more restrictive constraints which, in turn, cause more data loss in the rectification. For each format, we incrementally increase the size of the training input set and record the data loss percentage on the test inputs. At each step, we increase the size of training input by 200. Figure 11 presents the curves of the average data loss percentage of the test inputs of the different formats as the sizes of the training input sets change.

As expected, the curves initially drop rapidly, then approach a limit as the training set sizes become large. Note that the PNG and WAV curves converge more rapidly than the TIFF and JPEG curves. We attribute this phenomenon to the fact that the PNG and WAV formats are simpler than the TIFF and JPEG formats (see

Figure 10 for the number of semantically distinct fields of each format).

4.11.4 Overhead

We next evaluate the overhead introduced by SOAP. Figure 10 presents the average running time of the SOAP rectifier for processing the test inputs of each file format. All times are measured on an Intel 3.33GHz 6-core machine with SOAP running on only one core.

The results show that the majority of the execution time is incurred in the Hachoir parsing library, with the execution time per field roughly constant across the input file formats (so SWF files take longer to parse because they have significantly more fields than other kinds of files). We believe that users will find these rectification overheads negligible if not imperceptible during interactive use.

5 Sound Input Filter Generation

Many security exploits target software errors in deployed applications. One general approach to nullifying vulnerabilities is to deploy input filters that discard inputs that may trigger the errors.

Many previous filter generation systems are reactive [29, 30, 32, 48, 58] — they start with an observed exploit that targets a specific vulnerability, and then analyze the path to that vulnerability to obtain a filter that discards potentially malicious inputs that may exploit that path (and, in some cases, related paths that may lead to the vulnerability). Drawbacks include incomplete coverage (these techniques typically leave some paths to the vulnerability uncovered) and systems that are not protected until they are attacked.

We present a new proactive system, SIFT, for generating filters that discard inputs that may cause integer overflow errors at memory allocation and block copy sites. Unlike previous reactive systems, SIFT proactively analyzes the program before it executes to generate filters that take all execution paths into consideration. SIFT can therefore nullify exploits that target unknown vulnerabilities (i.e., zero-day attacks).

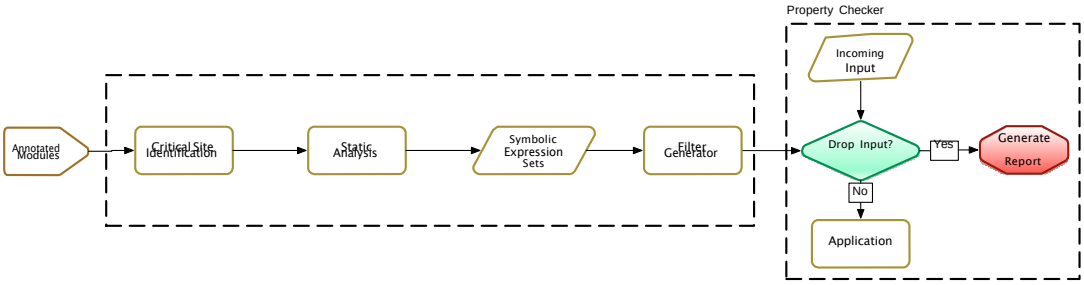


Figure 12: The SIFT architecture.

5.1 Static Analysis

The core of our technique is an interprocedural, demand-driven, backward static analysis that, given an integer expression e at a specified program point, propagates the expression backwards against the control flow until it has computed a symbolic expression set that includes all expressions that the application may

evaluate (in any execution) to obtain the value of e. The variables in these expressions represent the values of input fields. In effect, the symbolic expression set captures all of the possible computations that the program may perform on the input values to obtain the value of e.

A key challenge is that, to successfully extract effective symbolic expression sets, the analysis must reason precisely about interprocedural computations that use pointers to compute and manipulate values derived from input fields. Our analysis meets this challenge by deploying a novel combination of techniques including 1) a novel interprocedural weakest precondition analysis that works with symbolic representations of input fields and values accessed via pointers (including input fields read in loops and values accessed via pointers in loops), 2) a symbolic expression normalization algorithm that enables our loop invariant inference algorithm to successfully analyze loops that manipulate values derived from input fields or pointers, and 3) an alias analysis that ensures that the derived symbolic expressions correctly characterize the values that the program computes (including values stored in one procedure, then loaded in another procedure).

As is standard in the field, the alias analysis is designed to work with programs that do not access uninitialized or out of bounds memory. Our analysis therefore comes with the following soundness guarantee. If an input passes the filter for a given critical expression e, the input field annotations are correct (see Section 7.3), and the program has not yet accessed uninitialized or out of bounds memory when the program evaluates e, then no integer overflow occurs during the evaluation of e (including the evaluations of intermediate expressions that contribute to the final value of e).

5.2 SIFT Usage Model

Figure 12 presents the architecture of SIFT. This architecture is designed to support the following usage model:

Module Identification. Starting with an application that is designed to process inputs presented in one or more input formats, the developer identifies the modules within the application that process inputs of interest. SIFT will analyze these modules to generate an input filter for the inputs that these modules process.

Input Statement Annotation. The developer annotates the relevant input statements in the source code of the modules to identify the input field that each input statement reads.

Critical Site Identification. SIFT scans the modules to find all *critical sites* (currently, memory allocation and block copy sites). Each critical site has a *critical expression* that determines the size of the allocated or copied block of memory. The generated input filter will discard inputs that may trigger an integer overflow error during the computation of the value of the critical expression.

Static Analysis. For each critical expression, SIFT uses a demand-driven backwards static program analysis to automatically derive the corresponding *symbolic expression set*. Each expression in this set specifies, as a function of the input fields, how the value of the critical expression is computed along one of the program paths to the corresponding critical site.

Input Parser Acquisition. The developer obtains (typically from open-source parser repositories such as Hachoir [8]) a parser for the desired input format. This parser groups the input bit stream into input fields, then makes these fields available via a standard API.

Filter Generation. SIFT uses the input parser and symbolic expression sets to automatically generate the input filter. When presented with an input, the filter reads the fields of the input and, for each symbolic expression, determines if an integer overflow may occur when the expression is evaluated. If so, the filter discards the input. Otherwise, it passes the input along to the application.

The generated filters can be deployed anywhere along the network path from the input source to the application that ultimately processes the input.

5.3 Experimental Results

We used SIFT to generate input filters for modules in five real-world applications: VLC 0.8.6h [21] (a network media player), Dillo 2.1 [4] (a lightweight web browser), Swfdec 0.5.5 [18] (a flash video player), Swftools 0.9.1 [19] (SWF manipulation and generation utilities), and GIMP 2.8.0 [6] (an image manipulation application). Together, the analyzed modules contain 58 critical memory allocation and block copy sites. SIFT successfully generated filters for 52 of these 58 critical sites (SIFT's static analysis was unable to derive symbolic

expression sets for the remaining six critical sites, see Section 5.8.2 for more details). These applications contain six integer overflow vulnerabilities at their critical sites. SIFT's filters nullify all of these vulnerabilities.

Analysis and Filter Generation Time. We configured SIFT to analyze all critical sites in the analyzed modules, then generate a single, high-performance composite filter that checks for integer overflow errors at all of the sites. The maximum time required to analyze all of the sites and generate the composite filter was less than a second for each benchmark application.

False Positive Evaluation. We used a web crawler to obtain a set of least 6000 real-world inputs for each application (for a total of 62895 input files). We found no false positives — the corresponding composite filters accept all of the input files in this test set.

Filter Performance. We measured the composite filter execution time for each of the 62895 input files in our test set. The average time required to read and filter each input was at most 16 milliseconds, with this time dominated by the time required to read in the input file.

5.4 Contributions

This paper makes the following contributions:

- **SIFT:** We present SIFT, a proactive filter generation system for nullifying integer overflow vulnerabilities. SIFT scans modules to find critical memory allocation and block copy sites, statically analyzes the code to automatically derive symbolic expression sets that characterize how the application may compute the sizes of the allocated or copied memory blocks, and generates input filters that discard inputs that may trigger integer overflow errors in the evaluation of these expressions.
- **Static Analysis:** We present a new static analysis that automatically derives symbolic expressions that specify, as a function of the input fields, how the values of critical expressions are computed along the various possible execution paths to the corresponding critical site.
- **Experimental Results:** We present experimental results for SIFT on modules from five applications (VLC 0.8.6h, Dillo 2.1, Swfdec 0.5.5, Swftools 0.9.1, and GIMP 2.8.0). SIFT generates input filters that nullify integer

overflow vulnerabilities that may occur at 52 of the 58 memory allocation or block copy sites, including six known integer overflow errors. The filters exhibit no false positives when applied to 62895 real-world inputs downloaded from various sources on the Internet. The analysis and composite filter generation times are all less than a second. The composite filters execute in at most an average of 16 milliseconds per input, with the majority of the time devoted to reading in the input.

These contributions enable SIFT to proactively generate and deploy efficient input filters that nullify potentially unknown integer overflow vulnerabilities. We focus on memory allocation and block copy sites because these sites are often the target of attacks (in part because integer overflow errors at these sites often enable subsequent buffer overflow/code injection attacks).

5.5 Example

We next present an example that illustrates how siftname nullifies an integer overflow vulnerability in Swfdec 0.5.5, an open source shockwave flash player.

Figure 13 presents (simplified) source code from Swfdec. When Swfdec opens an SWF file with embedded JPEG images, it calls jpeg_decoder_decode() (line 1 in Figure 13) to decode each JPEG image in the file. This function in turn calls the function jpeg_decoder_start_of_frame() (line 7) to read the image metadata and the function jpeg_decoder_init_decoder() (line 22) to allocate memory buffers for the JPEG image.

There is an integer overflow vulnerability at lines 43-47 where Swfdec calculates the size of the buffer for a JPEG image as:

```
rowstride * (dec->height_block * 8 * max_v_sample /
dec->components[i].v_subsample)
```

At this program point, rowstride equals:

```
(jpeg_width + 8 * max_h_sample - 1) / (8 * max_h_sample)
* 8 * max_h_sample / (max_h_sample / h_sample)
```

while the rest of the expression equals

```
(jpeg_height + 8 * max_v_sample - 1) / (8 * max_v_sample)
* 8 * max_v_sample / (max_v_sample / v_sample)
```

```
1  int jpeg_decoder_decode(JpegDecoder *dec) {
2    ...
3    jpeg_decoder_start_of_frame(dec, ...);
4    jpeg_decoder_init_decoder (dec);
5    ...
6  }
7  void jpeg_decoder_start_of_frame(JpegDecoder*dec){
8    ...
9    dec->height = jpeg_bits_get_u16_be (bits);
10   /* dec->height = SIFT_input("jpeg_height", 16);*/
11   dec->width = jpeg_bits_get_u16_be (bits);
12   /* dec->width = SIFT_input("jpeg_width", 16); */
13   for (i = 0; i < dec->n_components; i++) {
14     dec->components[i].h_sample =getbits(bits, 4);
15     /* dec->components[i].h_sample =
16          SIFT_input("h_sample", 4); */
17     dec->components[i].v_sample =getbits(bits, 4);
18     /* dec->components[i].v_sample =
19          SIFT_input("v_sample", 4); */
20   }
21 }
22 void jpeg_decoder_init_decoder(JpegDecoder*dec){
23   int max_h_sample = 0;
24   int max_v_sample = 0;
25   int i;
26   for (i=0; i < dec->n_components; i++) {
27     max_h_sample = MAX(max_h_sample,
28       dec->components[i].h_sample);
29     max_v_sample = MAX(max_v_sample,
30       dec->components[i].v_sample);
31   }
32   dec->width_blocks=(dec->width+8*max_h_sample-1)
33     / (8*max_h_sample);
34   dec->height_blocks=(dec->height+8*max_v_sample-1)
35     / (8*max_v_sample);
36   for (i = 0; i < dec->n_components; i++) {
37     int rowstride;
38     int image_size;
39     dec->components[i].h_subsample=max_h_sample /
40      dec->components[i].h_sample;
41     dec->components[i].v_subsample=max_v_sample /
42      dec->components[i].v_sample;
43     rowstride=dec->width_blocks * 8 * max_h_sample /
44      dec->components[i].h_subsample;
45     image_size=rowstride * (dec->height_blocks * 8 *
46      max_v_sample / dec->components[i].v_subsample);
47     dec->components[i].image = malloc (image_size);
48   }
49 }
```

Figure 13: Simplified Swfdec source code. Input statement annotations appear in comments.

where jpeg_height is the 16-bit height input field value that Swfdec reads at line 9 and jpeg_width is the 16-bit width input field value that Swfdec reads at line 11. h_sample is one of the horizontal sampling factor values that Swfdec reads at line 14, while max_h_sample is the maximum horizontal sampling factor value. v_sample is one of the vertical sampling factor values that Swfdec reads at line 17, while max_v_sample is the maximum vertical sampling factor value. Malicious inputs with specifically crafted values in these input fields can cause the image buffer size calculation to overflow. In this case Swfdec allocates an image buffer that is smaller than required and eventually writes beyond the end of the allocated buffer.

$$\mathbf{S}: \quad \{((sext(jpeg\_width^{[16]},32)+8^{[32]}\times sext(h\_sample1^{[4]},32)-1^{[32]})/(8^{[32]}\times sext(h\_sample1^{[4]},32\times 8^{[32]}\times sext(h\_sample1^{[4]},32))/(sext(h\_sample1^{[4]},32)/sext(h\_sample2^{[4]},32))\times((sext(jpeg\_height^{[16]},32)+8^{[32]}\times sext(v\_sample1^{[4]},32)-1^{[32]})/(8^{[32]}\times sext(v\_sample1^{[4]},32\times 8^{[32]}\times sext(v\_sample1^{[4]},32))/(sext(v\_sample1^{[4]},32)/sext(v\_sample2^{[4]},32))\}$$

Figure 14: The symbolic expression set **S** for the Swfdec example. Each expression of **S** is a bit vector expression. The superscript indicates the bit width of each expression atom. "*sext*(*v*, *w*)" is the signed extension operation that transforms the value *v* to the bit width *w*.

The loop at lines 13-20 reads an array of horizontal and vertical factor values. Swfdec computes the maximum values of these factors in the loop at lines 26-31. It then uses these values to compute the size of the allocated buffer at each iteration in the loop (lines 36-48).

Analysis Challenges: This example highlights several challenges that SIFT must overcome to successfully analyze and generate a filter for this program. First, the expression for the size of the buffer uses pointers to access values derived from input fields. To overcome this challenge, SIFT uses an alias analysis [40] to reason precisely about expressions with pointers.

Second, the memory allocation site (line 47) occurs in a loop, with the size expression referencing input values read in a different loop (lines 13-19). Different instances of the same input field (h_sample and v_sample) are used to compute (potentially different) sizes for different blocks of memory allocated at the same site. To reason precisely about these different instances, the analysis works with an abstraction that materializes, on demand, abstract representatives of accessed input field and computed values (see Section 5.6). To successfully analyze the loop, the analysis uses a new loop invariant synthesis algorithm (which exploits a new expression normalization technique to reach a fixed point).

Finally, Swfdec reads the input fields (lines 14 and 17) and computes the size of the allocated memory block (lines 45-46) in different procedures. SIFT therefore uses an interprocedural analysis that propagates symbolic expressions across procedure boundaries to obtain precise symbolic expression sets.

We next describe how
siftname generates a sound input filter to nullify this integer overflow error.

Source Code Annotations: SIFT provides a declarative specification interface that enables the developer to specify which statements read which input fields. In this example, the developer specifies that the application reads the input fields jpeg_height, jpeg_width, h_sample, and v_sample at lines 10, 12, 15-16, and 18-19 in Figure 13. SIFT uses this specification to map the variables dec->height, dec->width, dec->components[i].h_sample, and dec->components[i].v_sample at lines 9, 11, 14, and 17 to the corresponding input field values. The field names h_sample and v_sample map to two arrays of input fields that Swfdec reads in the loop at lines 14 and 17.

Compute Symbolic Expression Set: SIFT uses a demand-driven, interprocedural, backward static analysis to compute the symbolic expression set **S** in Figure 14. **S** enumerates all of the expressions that Swfdec may evaluate, in any execution, to obtain the size of the allocated buffer (lines 45-46). Each expression is in *bit vector expression* form so that the expression accurately reflects the representation of the numbers inside the computer as fixed-length bit vectors as well as the semantics of arithmetic and logical operations as implemented inside the computer on these bit vectors.

In Figure 14, the superscripts indicate the bit width of each expression atom.

sext(*v*, *w*) is the signed extension operation that transforms the value *v* to the bit width *w*. SIFT also tracks the sign of each arithmetic operation in **S**. For simplicity, Figure 14 omits this information. SIFT soundly handles the loops that access the input field arrays h_sample and v_sample. The generated **S** reflects the fact that the variable dec->components[i].h_sample and the variable max_h_sample might be two different elements in the input array h_sample. In **S**, *h_sample*1 corresponds to max_h_sample and *h_sample*2 corresponds to dec->components[i].h_sample. SIFT handles v_sample similarly.

S includes all intermediate expressions evaluated at lines 32-35 and 39-46. In this example, **S** contains only a single expression. However, if there may be multiple execution paths, SIFT generates a symbolic expression set **S** with multiple expressions that cover all paths.

Generate Input Filter: Starting with the symbolic expression set **S**, SIFT generates an input filter that discards any input that may trigger an integer overflow when evaluating any expression in **S** (including all subexpressions). The generated filter extracts all instances of the input fields jpeg_height, jpeg_width, h_sample, and v_sample (these are the input fields that appear in **S**) from an incoming input. It then iterates over all combinations of pairs of the input fields h_sample and v_sample to consider all possible bindings of *h_sample*1,

$func$:= $proc(a_1, \ldots, a_k)$ { s; *return* v_{ret}; }
s := v = read(f) | v = c | v = v_1 op v_2 |
if (v) s_1 else s_2 | while (v) {s_1} |
s_1; s_2 | v = * p <label> |
* p = v <label> | v = call *proc* v_1 ... v_k

$f \in$ *Fields* $v, v_i, a_i \in$ *Vars*
$p \in$ *Pointers* $c \in$ *Int*

Figure 15: The Core Programming Language

*h_sample*2, *v_sample*1, and *v_sample*2 in **S**. For each binding, it checks the entire evaluation of **S** (including the evaluation of all subexpressions) for overflow. If there is no overflow in any evaluation, the filter accepts the input, otherwise it rejects the input.

5.6 Static Analysis

Core Language: Figure 15 presents the core language that we use to present the analysis. As is standard in the field, the analysis runs after the program analysis infrastructure (SIFT uses LLVM [12]) has lowered the program representation so that 1) nested expressions are converted into sequences of statements of the form $v = v_1$ op v_2 (where v, v_1, and v_2 are either non-aliased variables or LLVM-generated temporaries) and 2) all accesses to potentially aliased memory locations occur in load or store statements of the form v = * p <label> or * p = v <label> (the analysis uses the labels to materialize abstract representatives for accessed memory locations).

A statement of the form "v = read(f)" reads a value from an input field f. Because the input may contain multiple instances of the field f, different executions of the statement may return different values. For example, the loop at lines 14-17 in Figure 13 reads multiple instances of the h_sample and v_sample input fields.

Because it works with a lowered representation, our static analysis starts with a variable v at a critical program point. It then propagates v backward against the flow of control, first within the procedure that contains the critical program point, then up the call graph to the program entry point. In this way the analysis

op	:=	$+ \mid - \mid * \mid / \mid ...$
id	:=	$1 \mid 2 \mid 3 \mid 4 \mid ...$
Atom	:=	$c \mid v \mid f(id) \mid label(id)$
Expr	:=	$Atom \mid Expr\ \text{op}\ Expr$
ExprSet	≡	2^{Expr}

Figure 16: Symbolic Expression Sets

Statement s	Rules
$v = c$	$WP(v = c, \mathbf{S}) = \mathbf{S}[c/v]$
$v = v_1 op v_2$	$WP(v = v_1 op v_2, \mathbf{S}) = \mathbf{S}[v_1 op v_2/v]$
$v = read(f)$	$WP(v = read(f), \mathbf{S}) = \mathbf{S}[f(id)/v]$, $f(id)$ is fresh.
$s_1; s_2$	$WP(s_1; s_2, \mathbf{S}) = WP(s_1, WP(s_2, \mathbf{S}))$
$if\ (v)\ s_1\ else s_2$	$WP(if\ (v)\ s_1\ else s_2, \mathbf{S}) = WP(s_1, \mathbf{S}) \cup\ WP(s_2, \mathbf{S})$
$v = *\ p < label >$	$WP(v = *\ p < label >, \mathbf{S}) = \mathbf{S}[label(id)/v]$, $label(id)$ is fresh
$*\ p = v < label >$	$F(*\ p = v < label >, \mathbf{S}) = \mathbf{S}(v, label, label_1(id_1))(v, label, label_2(id_2)) \ldots (v, label, label_n(id_n))$ for all $label_1(id_1), \ldots, label_n(id_n)$ appearing in $\mathbf{S}$, where

$$\mathbf{S}(v, label, label_i(id_i)) = \begin{cases} \mathbf{S} & no\_alias(label, label_i) \\ \mathbf{S}[v/label_i(id_i)] \cup \mathbf{S} & may\_alias(label, label_i) \\ \mathbf{S}[v/label_i(id_i))] & must\_alias(label, label_i) \end{cases}$$

Figure 17: Weakest precondition analysis rules. The notation $\mathbf{S}[e_a/e_b]$ denotes the symbolic expression set obtained by replacing every occurrence of e_b in $\mathbf{S}$ with e_a.

computes an input expression set that soundly approximates how the program, starting with input field values f, may compute the value of v at the critical pro- gram point. The generated filters use the analysis results to check whether the input may trigger an integer overflow error in any of these computations.

Symbolic Expression Sets: Figure 16 presents the definition of *symbolic expres- sion sets*. There are four kinds of atoms: c represents a constant, v represents the variable v, $f(id)$ represents the value of an input field f (the analysis uses the natural number id to distinguish different instances of f), and $label(id)$ repre- sents a value returned by a load statement with the label $label$ (the analysis uses the natural number id to distinguish values loaded at different executions of the load statement).

Weakest Precondition Framework: Given a series of statements s and a symbolic expression set $\mathbf{S} \in ExprSet$, our analysis uses a weakest precondition analysis to compute a symbolic expression set $WP(s, \mathbf{S})$. The analysis ensures that if for all symbolic expressions $e \in WP(s, \mathbf{S})$, the evaluation of e at the program point before s does not encounter an integer overflow error, then the evaluation of any expression in **S** at the program point after s also does not encounter an integer overflow error. In contrast to many program analyses, which propagate information forward with the flow of control, this weakest precondition analysis propagates information backwards against the flow of control.

Analysis of Assignment, Conditional, and Sequence Statements: Figure 37 presents the analysis rules for basic program statements. The analysis of assignment statements replaces the assigned variable v with the assigned value (c, v_1 op v_2, or $f(id)$, depending on the assignment statement). Here the notation $\mathbf{S}[e_a/e_b]$ denotes the new expression set obtained by replacing every occurrence of e_b in **S** with e_a. The analysis rule for input read statements materializes a new id to represent the read value $f(id)$. This mechanism enables the analysis to correctly distinguish different instances of the same input field (because different instances have different ids).

The analysis of conditional statements takes the union of the symbolic expression sets from the analysis of the true and false branches of the if statement. The resulting symbolic expression set correctly takes the execution of both branches into account. The analysis of sequences of statements propagates the symbolic expression set backwards through the statements in sequence.

Analysis of Load and Store Statements: The analysis of a load statement $v = *\ p\ <label>$ replaces the assigned variable v with a materialized abstract value $label(id)$ that represents the loaded value. As for input read statements, the analysis uses a newly materialized id to distinguish values read on different executions of the load statement.

The analysis of a store statement $*\ p = v\ <label>$ uses the alias analysis to appropriately match the stored value v against all loads that may return that value. Specifically, the analysis locates all $label_i(id_i)$ atoms in **S** that either may or must load the value v that the store statement stores into the location p. If the alias analysis determines that the $label_i(id_i)$ expression must load v (i.e., the corresponding load statement will always access the value that the store statement stored into location p), then the analysis of the store statement replaces all occurrences of $label_i(id_i)$ with v.

If the alias analysis determines that the $label_i(id_i)$ expression may load v (i.e., on some executions the corresponding load statement may load v, on others it may not), then the analysis produces two symbolic expression sets: one with $label_i(id_i)$ replaced by v (for executions in which the load statement loads v) and one that leaves $label_i(id_i)$ in place (for executions in which the load statement loads a value other than v).

We note that, if the pointer analysis is imprecise, the expression sets may become intractably large. SIFT uses the DSA algorithm [40], a context-sensitive, unification-based pointer analysis. We found that, in practice, this analysis is precise enough to enable SIFT to efficiently analyze our benchmark applications (see Figure 21 in Section 5.8.2).

Analysis of Loop Statements: The loop analysis uses fixed-point iteration to discover an appropriate loop invariant that correctly summarizes the effect of the loop (regardless of the number of iterations that it may perform). Specifically, the analysis of a statement while (v) {s} computes a sequence of symbolic expression sets $\mathbf{S}_i$, where $\mathbf{S}_0 = \emptyset$ and $\mathbf{S}_i = norm(WP(s, S \cup S_{i-1}))$. Conceptually, each successive symbolic expression set $\mathbf{S}_i$ captures the effect of executing an additional loop iteration. The analysis terminates when it reaches a fixed point (i.e., when it has performed n iterations such that $S_n = S_{n-1}$). Here S_n is the discovered loop invariant.

The loop analysis normalizes the analysis result $WP(s, S \cup S_{i-1})$ after each iter- ation. For a symbolic expression set $\mathbf{S} = \{e_1, \ldots, e_n\}$, the normalization of $\mathbf{S}$ is $norm(\mathbf{S}) = \{norm(e_1), \ldots, norm(e_n)\}$, where $norm(e_i)$ is the normalization of each individual expression in $\mathbf{S}$ (using the algorithm presented in Figure 18).

Normalization facilitates loop invariant discovery for loops that read input fields or load values via pointers. Each analysis of the loop body during the fixed point computation produces new materialized values $f(id)$ and $label(id)$ with fresh ids. The new materialized $f(id)$ represent input fields that the current loop iteration reads; the new materialized $label(id)$ represent values that the current loop iteration loads via pointers. The normalization algorithm appropriately renumbers these ids in the new symbolic expression set so that the first appearence of each id is in lexicographic order. This normalization enables the analysis to recognize loop invariants that show up as equivalent successive analysis results that differ only in the materialized ids that they use to represent input fields and values accessed via pointers.

```
1  Input: Expression e
2  Output: Normalized expression e_norm
3
4  e_norm ← e
5  field_cnt ← {all → 0}
6  label_cnt ← {all → 0}
7  for a in Atoms(e) do
8    if a is in form f(id) then
9      nextid ← field_cnt(f) + 1
10     field_cnt ← field_cnt[f → nextid]
11     e_norm ← e_norm[* f(nextid/f(id)]
12   else if a is in form label(id) then
13     nextid ← label_cnt(label) + 1
14     label_cnt ← label_cnt[label → nextid]
15     e_norm ← e_norm[* label(nextid)/label(id)]
16   end if
17 end
18 for a in Atoms(e_norm) do
19   if a is in form * f(id) then
20     e_norm ← e_norm[f(id)/* f(id)]
21   else if a is in form * label(id) then
22     e_norm ← e_norm[label(id)/* label(id)]
23   end if
24 end
```

Figure 18: Normalization function *norm*(*e*). *Atom*(*e*) iterates over the atoms in the expression *e* from left to right.

The above algorithm will reach a fixed point and terminate if it computes the symbolic expression set of a value that depends on at most a statically fixed number of values from the loop iterations. For example, our algorithm is able to compute the symbolic expression set of the size parameter value of the memory allocation in Figure 13 — the value of this size parameter depends only on the values of jpeg_width and jpeg_height, the current values of h_sample and v_sample, and the maximum values of h_sample and v_sample, each of which comes from one previous iteration of the loop at line 26-31.

Note that the algorithm will not reach a fixed point if it attempts to compute a

symbolic expression set that contains an unbounded number of values from different loop iterations. For example, the algorithm will not reach a fixed point if it attempts to compute a symbolic expression set for the sum of a set of numbers computed within the loop (the sum depends on values from all loop iterations). To ensure termination, our current implemented algorithm terminates the analysis and fails to generate a symbolic expression set **S** if it fails to reach a fixed point after ten iterations.

In practice, we expect many programs may contain expressions whose values depend on an unbounded number of values from different loop iterations. Our analysis can successfully analyze such programs because it is demand driven — it only attempts to obtain precise symbolic representations of expressions that may contribute to the values of expressions in the analyzed symbolic execution set **S** (which, in our current system, are ultimately derived from expressions that appear at memory allocation and block copy sites). Our experimental results indicate that our approach is, in practice, effective for this set of expressions, specifically because these expressions tend to depend on at most a fixed number of values from loop iterations.

Analyzing Procedure Calls: We next present the interprocedural analysis for procedure call sites. Given a symbolic expression set **S** and a function call statement $v = call\ proc\ v_1 \ldots v_k$ that invokes a procedure $proc(a_1, a_2, \ldots, a_k)$ { s_b; $ret\ v_{ret}$ }, the analysis computes $WP(v = call\ proc\ v_1 \ldots v_k, \mathbf{S})$.[1]
Conceptually, the analysis performs two tasks. First, it replaces any occurrences of the procedure return value v in **S** (the symbolic expression set after the procedure call) with symbolic expressions that represent the values that the procedure may return. Second, it transforms **S** to reflect the effect of any store instructions that the procedure may execute. Specifically, the analysis finds expressions $label(id)$ in **S** that represent values that 1) the procedure may store into a location p that 2) the computation following the procedure may access via a load instruction that may access (a potentially aliased version of) p. It then replaces occurrences of $label(id)$ in **S** with symbolic expressions that represent the corresponding values computed (and stored into p) within the procedure.

Note that symbolic expressions derived from an analysis of the invoked procedure may contain occurrences of the formal parameters $a_1, \ldots, a_k$. The interpro-

[1] Note that because SIFT uses its underlying pointer analysis to disambiguate function pointers, it can analyze programs that invoke functions via function pointers.

```
1  Input: A symbolic expression set S
2  Output: WP(v = call proc v1 v2 ... vk, S),
3    where proc is defined as:
4    proc(a1, a2, ..., ak) { sb; ret vret }
5  Where: label1(id1), label2(id2), ..., labeln(idn)
6    are all atoms of the form label(id)
7    that appear in S.
8
9  R ← ∅
10 ST0 ←WP(sb, {vret})
11 for e0 in ST0[v1/a1]...[vn/an] do
12   ST1←WP(sb, {label1(id1)})
13   for e1 in ST1[v1/a1]...[vn/an] do
14     ...
15     STn←WP(sb, {labeln(idn)})
16     for en in STn[v1/a1]...[vn/an] do
17       e0^t ←make_fresh(e0)
18       ...
19       en^t ←make_fresh(en)
20       R←R ∪ S[e0^t/v]...[ei^t/labeli(idi)]...
21     end
22     ...
23   end
24 end
25 WP(v = call proc v1 v2 ... vk, S) ← R
```

Figure 19: Procedure Call Analysis Algorithm

cedural analysis translates these symbolic expressions into the name space of the caller by replacing occurrences of the formal parameters $a_1, \ldots, a_k$ with the corresponding actual parameters $v_1, \ldots, v_k$ from the call site.

Figure 19 presents the algorithm for analyzing procedure calls. At line 10 the algorithm analyzes the procedure body s_b to obtain a symbolic expression set $WP(s_b, \{v_{ret}\})$ representing the potential return values. It then translates this symbolic expression set into the name space of the caller by replacing the formal parameters $a_1, \ldots, a_n$ with the corresponding actual parameters $v_1, \ldots, v_n$. At lines 12 through 16 the algorithm analyzes the procedure body s_b for each representa-

tive $label_1(id_1), \ldots, label_n(id_n)$ that appears in **S** to derive a symbolic expression set that represents the set of values that the procedure may store into locations represented by the corresponding abstract materialized value $label_i(id_i)$.

At lines 11-16 the algorithm iterates over the derived symbolic expression sets. At line 20 it substitutes the derived expressions into **S** to compute the translated symbolic expression set $\mathbf{R} = WP(v = call\ proc\ v_1 \ldots v_k, \mathbf{S})$. To appropriately distinguish different invocations of the procedure, the analysis creates fresh versions of the $f(id)$ and $label(id)$ in the expressions $e_1, \ldots, e_n$ before it performs the substitution.

The algorithm avoids unnecessary reanalyses of the invoked procedure by caching the analysis results $WP(s_b, \{v_{ret}\})$ and $WP(s_b\{label_i(id_i)\})$ in a table for reuse at the analysis of other call sites that may invoke the procedure.

Propagation to Program Entry: To derive the final symbolic expression set at the start of the program, the analysis propagates the current symbolic expression set up the call tree through procedure calls until it reaches the start of the program. When the propagation reaches the entry of the current procedure *proc*, the algorithm uses the procedure call graph to find all call sites that may invoke *proc*.[2] It then propagates the current symbolic expression set **S** to the callers of *proc*, appropriately translating **S** into the naming context of the caller by substituting any formal parameters of *proc* that appear in **S** with the corresponding actual parameters from the call site. The analysis continues this propagation until it has traced out all paths in the call graph from the initial critical site where the analysis started to the program entry point. The final symbolic expression set is the union of the expression sets derived along all of these paths.

5.7 Implementation

We implemented SOAP in approximately 6000 lines of C++ code. We built the static analysis using the LLVM Compiler Infrastructure [12].

Analysis for C Programs: SOAP transforms the annotated application source code into the LLVM intermediate representation (IR) [12], scans the IR to identify critical values (i.e., size parameters of memory allocation and block copy

[2]Once again, because the analysis uses its pointer analysis to disambiguate function pointers, the call graph is accurate for call sites that use function pointers.

call sites) inside the developer specified module, and then performs the static analysis (see Section 5.6) for each identified critical value.

SOAP extends the analysis described in Section 5.6 to track the bit width of each expression atom. It also tracks the sign of each expression atom and arithmetic operation and correctly handles extension and truncation operations (i.e., signed extension, unsigned extension, and truncation) that change the width of a bit vector. SOAP therefore faithfully implements the representation of integer values in the C program.

By default, SIFT recognizes calls to standard C memory allocation routines (such as `malloc`, `calloc`, and `realloc`) and block copy routines (such as `memcpy`). SIFT can also be configured to recognize additional memory allocation and block copy routines (for example, `dMalloc` in Dillo).

SOAP provides a declarative specification language that developers use to indicate which input statements read which input fields. In our current implementation these statements appear in the source code in comments directly below the C statement that reads the input field. See lines 10, 12, 15-16, and 18-19 in Figure 13 for examples that illustrate the use of the specification language in the Swfdec example. The SIFT annotation generator scans the comments, finds the input specification statements, then inserts new nodes into the LLVM IR that contain the specified information. Formally, this information appears as procedure calls of the following form:

```
v = SIFT_Input("field_name", w);
```

where `v` is a program variable that holds the value of the input field with the field name `field_name`. The width (in bits) of the input field is `w`. The SIFT static analyzer recognizes such procedure calls as specifying the correspondence between input fields and program variables and applies the appropriate analysis rule for input read statements (see Figure 37).

The static analysis may encounter procedure calls (for example, calls to standard C library functions) for which the source code of the callee is not available. A standard way to handle this situation is to work with an annotated procedure declaration that gives the static analysis information that it can use to analyze calls to the procedure. If code for an invoked procedure is not available, by default SIFT currently synthesizes information that indicates that symbolic expressions are not available for the return value or for any values accessible (and therefore potentially stored) via procedure parameters (code following the procedure call

may load such values). This information enables the analysis to determine if the return value or values accessible via the procedure parameters may affect the analyzed symbolic expression set **S**. If so, SIFT does not generate a filter. Because SIFT is demand-driven, this mechanism enables SIFT to successfully analyze programs with library calls (all of our benchmark programs have such calls) as long as the calls do not affect the analyzed symbolic expressions.

We attribute any residual occurrences of abstract materialized values *label*(*id*) in the final symbolic expression set **S** to imprecision in the alias analysis (such values would correspond to accesses to uninitialized memory) and prune any expressions in **S** that contain such values.

Input Filter Generation: The filter operates as follows. It first uses an existing parser for the input format to parse the input and extract the input fields used in the input expression set **S**. Open source parsers are available for a wide of input file formats, including all of the formats in our experimental evaluation [8]. These parsers provide a standard API that enables clients to access the parsed input fields.

The generated filter evaluates each expression in **S** by replacing each symbolic input variable in the expression with the corresponding concrete value from the parsed input. If an integer overflow may occur in the evaluation of any expression in **S**, the filter discards the input and optionally raises an alarm. For input field arrays such as `h_sample` and `v_sample` in the Swfdec example (see Section 7.2), the input filter enumerates all possible combinations of concrete values. The filter discards the input if any combination can trigger the integer overflow error.

Given multiple symbolic expression sets generated from multiple critical program points, SOAP can create a single efficient filter that first parses the input, then checks the input against all symbolic expression sets in series on the parsed input. This approach amortizes the overhead of reading the input (in practice, reading the input consumes essentially all of the time required to execute the filter, see Figure 22) over all of the symbolic expression set checks.

5.8 Experimental Results

We evaluate SIFT on modules from five open source applications: VLC 0.8.6h [21] (a network media player), Dillo 2.1 [4] (a lightweight web browser),

Application	Distinct Fields	Relevant Fields
VLC	25	2
Dillo	47	3
Swfdec	219*	6
png2swf	47	4
jpeg2swf	300	2
GIMP	189	2

Figure 20: The number of distinct input fields and the number of relevant input fields for analyzed input formats. For Swfdec the second column shows the number of distinct fields in embedded JPEG images in collected SWF files.

Swfdec 0.5.5 [18] (a flash video player), Swftools 0.9.1 [19] (SWF manipulation and generation utilities), and GIMP 2.8.0 [6] (an image manipulation application). Each application uses a publicly available input format specification and contains at least one known integer overflow vulnerability (described in either the CVE database [2] or the Buzzfuzz paper [35]). All experiments were conducted on an Intel Xeon X5363 3.00GHz machine running Ubuntu 12.04.

5.8.1 Methodology

Input Format and Module Selection: For each application, we used SIFT to generate filters for the input format that triggers the known integer overflow vulnerability. We therefore ran SIFT on the module that processes inputs in that format. The generated filters nullify not only the known vulnerabilities, but also any integer overflow vulnerabilities at any of the 52 memory allocation or block copy sites in the modules for which SIFT was able to generate input expression sets (recall that there are 58 critical sites in these modules in total).

Input Statement Annotation: After selecting each module, we added annotations to identify the input statements that read relevant input fields (i.e., input fields that may affect the values of critical expressions at memory allocation or block copy sites). Figure 20 presents, for each module, the total number of distinct fields in our collected inputs for each format, the number of annotated input statements (in all of the modules the number of relevant fields equals the number of annotated input statements — each relevant field is read by a single input statement). We note that the number of relevant fields is significantly smaller

Application	Module	# of IR	Total	Input Relev	Inside Loop	Max Expr.	A
VLC	demux/wav.c	1.5k	5	3	0	2	
Dillo	png.c	39.1k	4	3	3	410	
Swfdec	jpeg/*.c	8.4k	22	19	2	144	
png2swf	all	11.0k	21	18	18	16	
jpeg2swf	all	2.5k	4	4	4	2	
GIMP	file-gif-load.c	3.2k	2	2	2	2	

Figure 21: Static Analysis and Filter Generation Results

than the total number of distinct fields (reflecting the fact that typically only a relatively small number of fields in each input format may affect the sizes of allocated or copied memory blocks).

The maximum amount of time required to annotate any module was approxi- mately half an hour (Swfdec). The total annotation time required to annotate all benchmarks, including Swfdec, was less than an hour. This annotation effort re- flects the fact that, in each input format, there are only a relatively small number of relevant input fields.

Filter Generation and Test: We next used SIFT to generate a single composite input filter for each analyzed module. We then downloaded at least 6000 real- world inputs for each input format on the web, and ran all of the downloaded inputs through the generated filters. There were no false positives (the filters accepted all of the inputs).

Vulnerability and Filter Confirmation: For each known integer overflow vul- nerability, we collected a test input that triggered the integer overflow. We con- firmed that each generated composite filter, as expected, discarded the input because it correctly recognized that the input would cause an integer overflow.

5.8.2 Analysis and Filter Evaluation

Figure 21 presents static analysis and filter generation results. This figure con- tains a row for each analyzed module. The first column (Application) presents the application name, the second column (Module) identifies the analyzed mod- ule within the application. The third column (# of IR) presents the number of analyzed statements in the LLVM intermediate representation. This number of

statements includes not only statements directly present in the module, but also statements from analyzed code in other modules invoked by the original module.

The fourth column (Total) presents the total number of memory allocation and block copy sites in the analyzed module. The fifth column (Input Relevant) presents the number of memory allocation and block copy sites in which the size of the allocated or copied block depends on the values of input fields. For these modules, the sizes at 49 of the 58 sites depend on the values of input fields. The sizes at the remaining nine sites are unconditionally safe — SIFT verifies that they depend only on constants embedded in the program (and that there is no overflow when the sizes are computed from these constants).

The sixth column (Inside Loop) presents the number of memory allocation and block copy sites in which the size parameter depends on variables that occurred inside loops. For these modules, the sizes at 29 of the 58 sites depend on loops relevant variables, for which SIFT needs to compute loop invariants to generate input filters.

The seventh column (Max Expr. Set Size) presents, for each application module, the maximum number of expressions in any expression set that occurs in the analysis of that module. The expression sets are reasonably compact (and more than compact enough to enable an efficient analysis) — the maximum expression set size over all modules is less than 500.

The final column (Analysis Time) presents the time required to analyze the module and generate a single composite filter for all of the successfully analyzed critical sites. The analysis times for all modules are less than a second.

SIFT is unable to generate symbolic expression sets **S** for six of the 58 call sites. For two of these sites (one in Swfdec and one in png2swf), the two expressions contain subexpressions whose value depends on an unbounded number of values from loop iterations. To analyze such expressions, our analysis currently requires an upper bound on the number of loop iterations. Such an upper bound could be provided, for example, by additional analysis or developer annotations. The remaining four expressions (two in png2swf and two in jpeg2swf) depend on the return value from `strlen()`. SIFT is not currently designed to analyze such expressions.

For each input format, we used a custom web crawler to locate and download at least 6000 inputs in that format. The web crawler starts from a Google search page for the file extension of the specific input format, then follows links in each

Application	Format	# of Input	Average Time
VLC	WAV	10976	3ms (3ms)
Dillo	PNG	18983	16ms (16ms)
Swfdec	SWF	7240	6ms (5ms)
png2swf	PNG	18983	16ms (16ms)
jpeg2swf	JPEG	6049	4ms (4ms)
GIMP	GIF	19647	9ms (9ms)

Figure 22: Generated Filter Results.

search result page to download files in the correct format.

Figure 22 presents, for each generated filter, the number of downloaded input files and the average time required to filter each input. We present the average times in the form Xms (Yms), where Xms is the average time required to filter an input and Yms is the average time required to read in the input (but not apply the integer overflow check). These data show that essentially all of the filter time is spent reading in the input.

5.8.3 Vulnerability Case Studies

In Section 7.2 we showed how SIFT handles the integer overflow vulnerability in Swfdec. We next investigate how SIFT handles the remaining five known vulnerabilities in our benchmark applications. Figure 24 presents the symbolic expression sets that SIFT generates for each of the five vulnerabilities in the analyzed modules.

VLC The VLC `wav.c` module contains an integer overflow vulnerability (CVE-2008-2430) when parsing WAV sound inputs. When VLC parses the format chunk of a WAV input, it first reads the input field *fmt_size*, which indicates the size of the format chunk. VLC then allocates a buffer to hold the format chunk. A large *fmt_size* field value (for example, 0xfffffffe) will cause an overflow to occur when VLC computes the buffer size.

We annotate the source code to specify where the module reads the *fmt_size* input field. SIFT then analyzes the module to obtain the input expression set **S**

(Figure 24), which soundly summarizes how VLC computes the buffer size from input fields.

```
1  // libpng main data process function.
2  void png_process_data(png_structp png_ptr,
3   png_infop info_ptr, ...) {
4    ...
5    while (png_ptr->buffer_size) {
6      // This is a wrapper for png_push_read_chunk
7      png_process_some_data(png_ptr, info_ptr);
8    }
9  }
10 // chunk handler dispatcher
11 void png_push_read_chunk(png_structp png_ptr,
12  png_infop info_ptr) {
13   if (!png_memcmp(png_ptr->chunk_name,png_IHDR,4)){
14     ...
15     png_handle_IHDR(png_ptr, info_ptr, ...);
16   }
17   ...
18   else if (!png_memcmp(png_ptr->chunk_name,
19    png_IDAT, 4)) {
20     ...
21     // Datainfo callback is called
22     png_push_have_info(png_ptr, info_ptr);
23     ...
24   }
25 }
26 #define PNG_ROWBYTES(pixel_bits,width)\
27   ((pixel_bits)>=8?\
28   ((width)*(((png_uint_32)(pixel_bits))>>3)):\
29   ((((width)*((png_uint_32)(pixel_bits)))+7)>>3))
30 void png_handle_IHDR(png_structp png_ptr,
31  png_infop info_ptr, ...) {
32   ...
33   // read individual png fields from input buffer
34   width = png_get_uint_31(png_ptr, buf);
35   /* width = SIFT_input("png_width", 32); */
36   height = png_get_uint_31(png_ptr, buf + 4);
37   /* height = SIFT_input("png_height", 32); */
38   bit_depth = buf[8];
39   /* bit_depth = SIFT_input("png_bitdepth", 8); */
40   ...
41   png_ptr->width = width;
42   png_ptr->height = height;
43   png_ptr->bit_depth = (png_byte)bit_depth;
44   ...
45   switch (png_ptr->color_type) {
46      case PNG_COLOR_TYPE_GRAY:
47      case PNG_COLOR_TYPE_PALETTE:
48         png_ptr->channels = 1;
49         break;
50      case PNG_COLOR_TYPE_RGB:
51         png_ptr->channels = 3;
52         break;
53      case PNG_COLOR_TYPE_GRAY_ALPHA:
54         png_ptr->channels = 2;
55         break;
56      case PNG_COLOR_TYPE_RGB_ALPHA:
57         png_ptr->channels = 4;
58         break;
59   }
60   png_ptr->pixel_depth = (png_byte)(
61    png_ptr->bit_depth * png_ptr->channels);
62   png_ptr->rowbytes = PNG_ROWBYTES(
63    png_ptr->pixel_depth, png_ptr->width);
64 }
65 // Dillo datainfo initialization callback
66 static void Png_datainfo_callback(png_structp png_ptr,
67  ...) {
68   DilloPng *png;
69   png = png_get_progressive_ptr(png_ptr);
70   ...
71   // where the overflow happens
72   png->image_data = (uchar_t *) dMalloc(
73     png->rowbytes * png->height);
74   ...
75 }
```

Figure 23: The simplified source code from Dillo and libpng with annotations inside comments.

Dillo Dillo contains an integer overflow vulnerability (CVE-2009-2294) in its png module. Figure 26 presents the simplified source code for this exam- ple. Dillo uses the libpng library to read PNG images. The libpng runtime calls png_process_data() (line 2) to process each PNG image. This function then

calls png_push_read_chunk() (line 11) to process each chunk in the PNG image. When the libpng runtime reads the first data chunk (the IDAT chunk), it calls the Dillo callback png_datainfo_callback() (lines 66-75) in the Dillo PNG processing module. There is an integer overflow vulnerability at line 73 where Dillo calculates the size of the image buffer as png->rowbytes*png->height. On a 32-bit machine, inputs with large width and height fields can cause the image buffer size calculation to overflow. In this case Dillo allocates an image buffer that is smaller than required and eventually writes beyond the end of the allocated buffer.

Figure 24 presents the input expression set **S** for Dillo. **S** soundly takes intermediate computations over all execution paths into consideration, including the switch branch at lines 45-59 that sets the variable png_ptr->channels and PNG_ROWBYTES macro at lines 26-29. Note that the constant $c^{[32]}$ in **S** corresponds to the possible values of png_ptr->channels, which are between 1 and 4.

Swftools Swftools is a set of utilities for creating and manipulating SWF files. Swftools contains two tools png2swf and jpeg2swf, which transform PNG and JPEG images to SWF files. Each of these two tools contains an integer overflow vulnerability (CVE-2010-1516).

When processing PNG images, Swftools calls getPNG() at png2swf.c:763 to read the PNG image into memory. getPNG() first calls png_read_header() to locate and read the header chunk which contains the PNG metadata. It then uses the metadata information to calculate the length of the image data at png.h:502. There is no bounds check on the width and the height value from the header chunk before this calculation. On a 32-bit machine, a PNG image with large width and height values will trigger the integer overflow error.

We annotate the statements that read input fields png_width and png_height and use SIFT to derive the input expression set for this vulnerability. Figure 24 presents the input expression set **S**.

jpeg2swf contains a similar integer overflow vulnerability when processing JPEG images. At jpeg2swf.c:171 jpeg2swf first calls the libjpeg API to read jpeg image. At jpeg2swf.c:173, jpeg2swf then immediately calculates the size of a memory buffer for holding the jpeg file in its own data structure. Because it directly uses the input width and height values in the calculation without range

checks, large width and height values may cause overflow errors. Figure 24 presents the symbolic expression set **S** for jpeg2swf.

GIMP GIMP contains an integer overflow vulnerability (CVE-2012-3481) in its GIF loading plugin file-gif-load.c. When GIMP opens a GIF file, it calls load_image at file-gif-load.c:335 to load the entire GIF file into memory. For each individual image in the GIF file, this function first reads the image metadata information, then calls ReadImage to process the image. At file-gif-load.c:1064, the plugin calculates the size of the image output buffer as a function of the product of the width and height values from the input. Because it uses these values directly without range checks, large height and width fields may cause an integer overflow. In this case GIMP may allocate a buffer smaller than the required size.

We annotate the source code based on the GIF specification and use SIFT to derive the input expression set for this vulnerability. Figure 24 presents the generated symbolic expression set **S**.

5.8.4 Discussion

The experimental results highlight the combination of properties that, together, enable SIFT to effectively nullifying potential integer overflow errors at memory allocation and block copy sites. SIFT is efficient enough to deploy in production on real-world modules (the combined program analysis and filter generation times are always under a second), the analysis is precise enough to successfully generate input filters for the majority of memory allocation and block copy sites, the results provide encouraging evidence that the generated filters are precise enough to have few or even no false positives in practice, and the filters execute efficiently enough to deploy with acceptable filtering overhead.

VLC $\{(fmt\_size^{[32]} + 1^{[32]}) + 2^{[32]},\ fmt\_size^{[32]} + 2^{[32]}\}$

png2swf $\{(c^{[32]} \times png\_width^{[32]}) \times png\_height^{[32]} + 65536^{[32]} \mid c = 1,2,3,4\}$

jpeg2swf $\{(jpeg\_width^{[32]} \times jpeg\_height^{[32]}) \times 4^{[32]}\}$

Dillo $\{((png\_width^{[32]} \times (c^{[32]} \times sext(png\_bitdepth^{[8]}, 32)) + 7^{[32]}) >> 3^{[32]}) \times png\_height^{[32]},\ png\_width^{[32]} \times ((c^{[32]} \times sext(png\_bitdepth^{[8]}, 32)) >> 3^{[32]}) \times png\_height^{[32]} \mid c^{[32]} = 1,2,3,4\}$

GIMP $\{(gif\_width^{[32]} \times gif\_height^{[32]}) \times 2^{[32]},\ gif\_width^{[32]} \times gif\_height^{[32]} \times 4^{[32]}\}$

Figure 24: The symbolic expression set **S** in the bit vector form for VLC, Swftools-png2swf, Swftools-jpeg2swf, Dillo and GIMP. The superscript indicates the bit width of each expression atom. "*sext*(*v*, *w*)" is the signed extension operation that transforms the value *v* to the bit width *w*.

6 DIODE

Integer overflow errors are an insidious source of software failures and security vulnerabilities [34, 57, 2]. Because programs with latent overflow errors often process typical inputs correctly, such errors can easily escape detection during testing only to appear later in production. Overflow errors that occur at memory allocation sites can be especially problematic as they comprise a prime target for code injection attacks. A typical scenario is that a malicious input exploits the overflow to cause the program to allocate a memory block that is too small to hold the data that the program will write into the allocated block. The resulting out-of-bounds writes can easily enable code injection attacks [34].

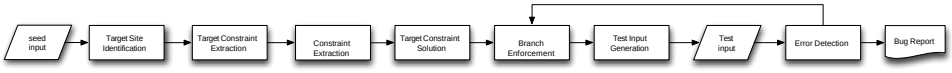


Figure 25: System Overview

6.1.1 DIODE

We present a new technique and system, DIODE (Directed Integer Overflow Discovery Engine), for automatically generating inputs that trigger integer overflow errors at critical sites. DIODE starts with a *target site* (such as a memory allocation site) and a *target value* (such as the size of the allocated memory block). It then uses symbolic execution to obtain an *target expression* that characterizes how the program computes the target value as a function of the input. It then transforms the target expression to obtain a *target constraint*. If the input 1) satisfies the target constraint while 2) causing the program to execute the target site, then it will trigger the error.

Sanity Checks: A key observation behind the design of DIODE is that programs often perform sanity checks on the input before they use the input to compute target values. If the input does not pass the sanity checks, the program typically emits an error or warning message and does not further process the input. To trigger an overflow, an input must therefore take the *same path* through the sanity checks as typical inputs that the program processes successfully.

One obvious way to obtain an input that satisfies the sanity checks is start with a *seed input* that causes one or more target sites to execute, then use a solver to obtain a new input that 1) satisfies the target constraint as well as 2) additional constraints that force the solver to generate an input that takes the *same path* to the target site as the seed input. This approach ensures that the input passes the sanity checks.

Blocking Checks: Unfortunately, our results indicate that this approach often fails because, in most cases, the path that the seed input takes through the computation contains additional *blocking checks* that prevent any input that satisfies these checks from triggering the error. To trigger an overflow, an input must take a *different path* through these blocking checks. The challenge is therefore to find inputs that 1) satisfy the target constraint, 2) satisfy the sanity checks, and 3) find a path through the blocking checks to execute the target site. DIODE meets this challenge as follows:

- **Target Site Identification:** Using a fine-grained dynamic taint analysis on the program running on the seed input, DIODE identifies all memory allocation sites that are influenced by values from the seed input. These sites are the *target sites*.

- **Target Constraint Extraction:** Based on instrumented executions of the program, DIODE extracts a symbolic target expression that characterizes how the program computes the target value (the size of the allocated memory block) at each target memory allocation site. The inputs that appear in this expression are the *relevant inputs*. Using the target expression, DIODE generates a *target constraint* that characterizes all inputs that would cause the computation of the target value to overflow (as long as the input also causes the program to compute the target value).
- **Branch Constraint Extraction:** Again based on instrumented executions of the program, DIODE extracts the sequence of conditional branch instructions that the program executes to generate the path to the target memory allocation site. To ensure that DIODE considers only relevant conditional branches, DIODE discards all branches whose condition is not influenced by relevant inputs.

  For each remaining conditional branch, DIODE generates a *branch constraint* that characterizes all input values that cause the execution to take the same path at that branch as the seed input. DIODE will use these branch constraints to generate candidate test inputs that force the program to follow the same path as the seed input at selected conditional branches.
- **Target Constraint Solution:** DIODE invokes the Z3 SMT solver [33] to obtain input values that satisfy the target constraint. If the program follows a path that evaluates the target expression at the target memory allocation site, DIODE has successfully generated an input that triggers the overflow. If the program performs no sanity checks on the generated values, this step typically delivers an input that triggers the overflow.
- **Goal-Directed Conditional Branch Enforcement:** If the previous step failed to deliver an input that triggers an overflow, DIODE compares the path that the seed input followed with the path that the generated input followed. These two paths must differ (otherwise the generated input would have triggered an overflow).

  DIODE then finds the first (in the program execution order) relevant conditional branch where the two paths diverge (i.e., where the generated input takes a different path than the seed input). We call this conditional branch the *first flipped branch*.

  DIODE adds the branch constraint from the first flipped branch to the

constraint that it passes to the solver, forcing the solver to generate a new input that takes the same path as the seed input at the that first flipped branch. DIODE then runs the program on this new generated input to see if it triggers the overflow.

DIODE continues this goal-directed branch enforcement algorithm, incrementally adding the branch constraints from first flipped branches, until either 1) it generates an input that triggers the overflow or 2) it generates an unsatisfiable constraint.

If the program does not contain relevant sanity checks, DIODE will typically find an input that triggers the overflow immediately when it solves the target constraint. If the program does contain relevant sanity checks, DIODE enforces flipped sanity checks in the order in which they are executed by the program. Each iteration of the goal-directed conditional branch enforcement algorithm forces the solver to produce an input that satisfies the next relevant unsatisfied sanity check.

As soon as DIODE enforces enough relevant sanity checks, it typically obtains an input that triggers the overflow (if such an input exists). Because the test inputs enforce only relevant branch conditions associated with previously failed relevant sanity checks, this approach gives the input the freedom it needs to navigate the blocking checks that would, if enforced, cause the program to fail to execute the target site (and therefore fail to generate an overflow).

6.1.2 Experimental Results

We evaluate DIODE on five applications: Dillo 2.1, VLC 08.6h, SwfPlay 0.5.5, CWebP 0.3.1, and ImageMagick 6.5.2. We start by using DIODE to locate the target memory allocation sites (there are 40 of these sites) and extract, for each site, the target constraint. The target constraint for 17 of the 40 target sites is unsatisfiable. For 9 of the remaining 23 target sites, DIODE was unable to generate an overflow-triggering input. Our manual inspection of the source code verified that the applications contain sanity checks that prevent any input from triggering an overflow at these target sites.

DIODE was able to generate inputs that trigger overflows at all of the remaining 14 sites. We were aware of 3 of these overflows prior to starting the study; the remaining 11 were new. We verified that at least 4 of the new overflow errors

are still present in the latest versions of these applications as of the submission date of this paper. For 2 of the 14 sites, DIODE was able to generate an overflow-triggering input with a constraint that forced the input to take the same path as the seed input. For the remaining 12 sites, the presence of relevant blocking checks requires any overflow-triggering input to take a different path to the target site.

For 9 of the 14 sites DIODE was able to generate an overflow-triggering input without enforcing any conditional branches. The remaining 5 sites require the enforcement of a minimum of 2, average of 4, and maximum of 5 conditional branches. Our manual inspection of the source code indicates that all of the enforced conditional branches involve sanity checks on relevant inputs (all but one of which were apparently not specifically designed to check for overflows). Our results also indicate that, if the application does perform relevant sanity checks and the input generation strategy does not take these checks into account, the input generation strategy is unlikely to find inputs that trigger an overflow even when such inputs exist (Section 7.4).

6.1.3 Engineering Challenges and Solutions

DIODE works directly on off-the-shelf, production stripped x86 binaries with no need for symbol information or source code. Given a binary and one or more seed inputs, DIODE executes instrumented versions of the binary to extract the symbolic target expressions and branch conditions for each target memory allocation site. For scalability reasons, DIODE stages the symbolic expression extraction as follows.

The first stage runs the application using fine-grained taint tracing to find memory allocation sites in which the input influences the size of the allocated memory block. This size is the target value of the site. This stage also obtains, for each target value, the *relevant input bytes*, i.e., the input bytes that influence the target value. The second stage runs the application again, recording a (compressed for efficiency) symbolic representation of each computation that the relevant input bytes influence. The third stage reads the symbolic representation of the computation to automatically derive the symbolic target expressions at the target memory allocation sites (these expressions capture the computation that the application performs on the relevant input bytes to obtain the target value) and the symbolic branch condition expressions at the relevant conditional

branches. This staging is essential in enabling DIODE to scale to real-world applications — attempting to record a symbolic representation of all computations that the application performs is clearly infeasible for real-world applications.

Given a seed input and candidate values from the Z3 SMT solver for relevant input fields within the seed input, DIODE uses Hachoir [8] and Peach [15] to generate a new input file with the candidate values. Together, Hachoir and Peach reconstruct the input file to accommodate the values, applying techniques such as checksum recalculation.

6.1.4 DIODE and Multi-Application Code Transfer

Once DIODE has identified the error, the next step is to eliminate the error. The standard approach is to report the error to the developers of the application, then wait for them to develop and distribute a patch [**?**]. Drawbacks of this approach include the patch development and distribution time and the difficulty of obtaining any patch at all if the application is no longer under development or maintained.

In response to this problem, we have developed CodePhage, an automatic code transfer system [54]. CodePhage starts with an input that exposes an error, a related input that the application processes correctly, and a donor application that processes both inputs correctly (such applications are typically readily available for standard input file formats). CodePhage automatically discovers code in the donor that eliminates the error, then transfers this code into the original application to eliminate the error. CodePhage operates directly on stripped x86 binary donors to generate source-level patches. The code transfer includes automatic data structure translation and the automatic location of appropriate code insertion points in the recipient. Combining CodePhage with DIODE produces a system that automatically discovers and eliminates integer overflow errors — DIODE generates inputs that expose errors; CodePhage uses these inputs to locate and transfer code from donor applications to eliminate the errors. To the best of our knowledge, CodePhage is the first system to automatically transfer code between applications.

6.1.5 Continuous Automatic Improvement

Given a the ability to automatically expose errors via tools such as DIODE and the ability to automatically repair these errors via tools such as CodePhage [54] (as well as the ability to automatically generate repairs using techniques such as ClearView [50], Error Virtualization [**?**, **?**], Failure-Oblivious Computing [**?**], and RCV [44]), the next step is to build *continuous automatic improvement* systems that automatically search for errors and generate patches that repair the encountered errors. ClearView's automatic patch generation capability provides continuous improvement driven by responses to attacks and errors that users encounter in production use [50]. Augmenting the ClearView continuous improvement approach with continuously executing automatic error detection tools would make it possible to detect and repair errors before users encounter them and before attackers can exploit them. The result would be significantly more secure and robust software systems.

6.1.6 Contributions

- **Targeted Input Generation:** It introduces the approach of automatically generating error-triggering inputs that target potentially vulnerable program sites.
- **Sanity and Blocking Checks:** It identifies sanity and blocking checks as an important challenge for techniques that aspire to discover error-triggering inputs. Critically, our results indicate that if the program contains relevant sanity checks, one way to identify relevant sanity checks and generate inputs that satisfy these checks is to incrementally find and enforce first flipped conditional branches.
- **DIODE:** It presents DIODE, an implemented system that works with programs that contain relevant sanity checks to automatically generate inputs that trigger overflow errors. Starting with seed inputs that execute a set of target memory allocation sites, DIODE uses (optimized) symbolic execution to obtain symbolic expressions that characterize how input values determine the path through the computation to the target site and control the target value (the number of bytes that the target site allocates).

  Using a targeted approach, DIODE generates a sequence of inputs, each

```
// libpng main data process function.
void png_process_data(png_structp png_ptr,
 png_infop info_ptr, ...) {
  ...
  while (png_ptr->buffer_size) {
    // This is a wrapper for png_push_read_chunk
    png_process_some_data(png_ptr, info_ptr);
  }
}
void png_push_read_chunk(png_structp png_ptr,
 png_infop info_ptr) {
  if (!png_memcmp(png_ptr->chunk_name,png_IHDR,4)){
    ...
    png_handle_IHDR(png_ptr, info_ptr, ...);
  }
  else if (!png_memcmp(png_ptr->chunk_name,png_IDAT, 4)) {
    // Datainfo callback is called
    png_push_have_info(png_ptr, info_ptr);
  }
}
png_check_IHDR(png_structp png_ptr,
   png_uint_32 width, png_uint_32 height, int bit_depth...) {
    ...
    //Check 3: Height < 1000000L
    if (height > PNG_USER_HEIGHT_MAX) {
        png_warning(png_ptr,
                  "Image width exceeds user limit in IHDR");
        error = 1;
    }
    //Check 4: Width < 1000000L
    if (width > PNG_USER_WIDTH_MAX) {
        png_warning(png_ptr,
                  "Image width exceeds user limit in IHDR");
        error = 1;
    }
}
png_get_uint_31(png_structp png_ptr, png_const_bytep buf) {
    png_uint_32 uval = png_get_uint_32(buf);
    // Checks 1 & 2: Checks that width/height < 0x7fffffffL
    if (uval > PNG_UINT_31_MAX)
        png_error(png_ptr,
                  "PNG unsigned integer out of range");
    return (uval);
}
#define PNG_ROWBYTES(pixel_bits,width) ((pixel_bits)>=8? \
  ((width)*(((png_uint_32)(pixel_bits))>>3)):\
  ((((width)*((png_uint_32)(pixel_bits)))+7)>>3))
void png_handle_IHDR(png_structp png_ptr,
 png_infop info_ptr, ...) {
  ...
  // read individual png fields from input buffer
  width     = png_get_uint_31(png_ptr, buf);
  height    = png_get_uint_31(png_ptr, buf + 4);
  bit_depth = buf[8];
  ...
  png_ptr->width        = width;
  png_ptr->height       = height;
  png_ptr->bit_depth    = (png_byte)bit_depth;
  ...
  png_ptr->pixel_depth = (png_byte)(
        png_ptr->bit_depth * png_ptr->channels);
  png_ptr->rowbytes = PNG_ROWBYTES(
        png_ptr->pixel_depth, png_ptr->width);
}
png_memset_check (png_structp png_ptr, png_voidp s1, int value,
          png_uint_32 length)
{
   png_size_t size;
   size = (png_size_t)length;
   if ((png_uint_32)size != length)
      png_error(png_ptr, "Overflow in png_memset_check.");
   return (png_memset (s1, value, size));
}
// Dillo datainfo initialization callback
static void
Png_datainfo_callback(png_structp png_ptr, png_infop info_ptr)
{
   DilloPng *png;
   ...
   // Check 5:  Incorrect check of max image size
   if (abs(png->width*png->height) > IMAGE_MAX_W * IMAGE_MAX_H) {
      MSG("suspicious image size request %ldx%ld\n",
          png->width, png->height);
      return;
   }
   // Where the overflow happens
   png->image_data = (uchar_t *)dMalloc(png->rowbytes * png->height);
}
```

Figure 26: Simplified source code from Dillo 2.1 and libpng

of which enforces the next relevant conditional branch to find and satisfy the sanity checks that would otherwise prevent the input from triggering the overflow at the target site. The goal is to find inputs that satisfy the relevant sanity checks while preserving the ability of the input to successfully traverse relevant blocking checks and reach the target site.

- **Experimental Results:** It presents experimental results that characterize the effectiveness of DIODE in discovering overflow errors. For our benchmark applications, DIODE discovers 14 overflows, 11 of which are new. For 9 of these overflows, DIODE generates overflows without enforcing any conditional branches. We attribute this success to a lack of relevant sanity checks in the program.
  For the remaining 5 overflows, DIODE discovers the overflow after enforcing a modest (2 to 5) number of conditional branches. We attribute this success to the ability of DIODE to 1) successfully identify and satisfy relevant sanity checks that appear in these programs while 2) preserving the ability of the input to traverse relevant blocking checks that would otherwise prevent the execution of the target site.

Fuzzing [15, 17] and concolic execution [47, 26, 36, 27] have been shown to be effective in discovering errors in the initial input parsing stages of computations, but have had little to no success in exposing errors that lie deep within the program. DIODE shows that discovering and targeting specific potentially vulnerable program sites can effec-tively expose such deep errors. One of the keys to success is new techniques that work appropriately with sanity and blocking checks to obtain inputs that can successfully tra-verse these obstacles to reach the target site. The success of DIODE in exposing integer overflow vulnerabilities opens up the field to the further development of other targeted techniques that work effectively with sanity and blocking checks to expose deep errors.

6.*1* Example

We next present an example that illustrates how DIODE automatically generates an input that triggers an integer overflow in Dillo 2.1, a lightweight open source web browser [4]. Figure 26 presents the simplified source code for this example. This code is from the libpng library, which Dillo uses to read PNG images.

Target Site Discovery: DIODE runs Dillo on the seed input, using a fine-grained dynamic taint analysis to track the propagation of input bytes through

the program. The libpng runtime calls `png_process_data()` (line 2) to process each PNG image. This function then calls `png_push_read_chunk()` (line 10) to process each chunk in the PNG image. When the libpng runtime reads the first data chunk (the IDAT chunk), it calls the Dillo callback `png_datainfo_callback()` (lines 76-88) in the Dillo PNG processing module. At line 87, Dillo invokes dMalloc() to allocate the image buffer. Because the size of the allocated memory block is influenced by the input, DIODE identifies the site as a *target memory allocation site*.

Dillo computes the size of the allocated image buffer as `png→rowbytes * png→height`. This is the *target value*. DIODE's goal is to generate an input that 1) executes the target site at line 87 and 2) causes the computation of the target value `png→rowbytes * png→height` to overflow. The taint information indicates that the target value is influenced by the PNG width, height, and bitdepth fields in the seed input file. These fields are the *relevant input bytes*.

Target Expression Extraction: Next, DIODE runs the application again, this time with additional instrumentation that records all calculations that involve the relevant input bytes. DIODE uses the recorded information to extract the symbolic *target expression*, which characterizes how the application computes the target value (recall that this target value is the size of the allocated image buffer) as a function of the input bytes. Conceptually, this expression is `((width*(4*bitdepth))>>3)*height`, where `width`, `bitdepth`, and `height` are the PNG width, bitdepth, and height fields in the input file. Large values of these fields will cause this expression to overflow. Because of endianness conversions that take place when Dillo reads in the input field values, the actual target expression is:

```
MallocArg(Mul(32,Mul(32,Add(32,ToSize(32,UShr(32,BvAnd(32,
HachField(32, '/header/width'),Constant(0xFF000000)),
Constant(24))),Add(32,Add(32,Shl(32,ToSize(32,
BvAnd(32,HachField(32, '/header/width'), Constant(0xFF))),
Constant(24)),Shl(32,ToSize(32,UShr(32,BvAnd(32,HachField(32,
'/header/width'),Constant(0xFF00)),Constant(8))),
Constant(16))),Shl(32,ToSize(32,UShr(32,BvAnd(32,
HachField(32, '/header/width'),Constant(0xFF0000)),
Constant(16))),Constant(8)))),ToSize(32,Shrink(8,
UShr(32,ToSize(32,Shrink(8,Mul(32,ToSize(32,
HachField(8, '/header/bit_depth')),Constant(4)))),
Constant(3))))),Add(32,ToSize(32,UShr(32,BvAnd(32,
HachField(32, '/header/height'),Constant(0xFF000000)),
Constant(24))),Add(32,Add(32,Shl(32,ToSize(32,
BvAnd(32,HachField(32, '/header/height'),
Constant(0xFF))),Constant(24)),Shl(32,ToSize(32,
UShr(32,BvAnd(32,HachField(32, '/header/height'),
```

```
Constant(0xFF00)),Constant(8))),Constant(16))),Shl(32,
ToSize(32,UShr(32,BvAnd(32,HachField(32,
'/header/height'),Constant(0xFF0000)),Constant(16))),
Constant(8))))),Constant(0xFFFFFFFF))
```

Here `TargetSite` indicates that, to overflow, the expression must be greater than the constant `0xFFFFFFFF` (at the end of the last line of the expression). The expression itself references the PNG width, bitdepth, and height fields from the input file as `/header/width`, `/header/bit_depth`, and `/header/height`. The remainder of the expression captures the computation of the target value as described above. It also incorporates constructs (such as `Shl` and `BvAnd`) that capture the conversion of the input values from big-endian to little-endian form. From this target expression, DIODE extracts a target constraint that is satisfied if and only if the computation of the target expression overflows. The variables in this target constraint represent the `/header/width`, `/header/bit_depth`, and `/header/height` PNG input file fields. The target constraint faithfully represents integer arithmetic as implemented in the hardware.

Target Constraint: DIODE next uses the Z3 solver [33] to obtain *candidate values* for the relevant input byte values that would cause the target value to overflow. In this example, the solution sets `/header/width` to 3880563055L, `/header/bit_depth` to 4, and `/header/height` to

689749785L. It then uses Hachoir [8] and Peach [15] to generate a new input file with the candidate values (we call this input file the *initial input file*) and executes Dillo on this new input file. Dillo and libpng contain sanity checks that together prevent the input from triggering the overflow.

Sanity Checks: Dillo and libpng collectively contain five sanity checks. The first two checks occur in `png_get_uint_31` (line 37), which checks that the PNG height and width values are less than 0x7fffffffL. The third and fourth sanity checks occur in `png_check_IHDR` (lines 21–36), which check that the PNG height and width values are less than one million. The fifth and final sanity check occurs at line 72, immediately before the target memory allocation site at line 87. This final sanity check attempts to ensure that the size of the allocated image does not exceed a specified value (`IMAGE_MAX_W * IMAGE_MAX_H`) (which is 6000 * 6000). This final check contains an overflow error that prevents it from recognizing and correctly rejecting some inputs that cause overflows at the target memory allocation site at line 87.

Symbolic Branch Condition Extraction: DIODE uses the recorded instrumentation information to extract symbolic expressions (the branch conditions) that characterize how the application computes the values of the branch conditions at conditional branch instructions that are directly influenced by the relevant input bytes. For Dillo, the extracted branch conditions characterize how Dillo computes the branch conditions for the sanity checks described above.

Blocking Checks: DIODE is capable of generating a constraint over the relevant input bytes that 1) cause the target value to overflow and 2) cause the application to follow the same path through the conditional branches to the target site as the seed input. If this constraint were satisfiable, DIODE could then use the solution to generate an input file that would trigger the overflow. This constraint is not satisfiable. Dillo and libpng contain *blocking checks* that prevent any input that would trigger an overflow from following the same path through the relevant branches to the target site.

The blocking checks occur in the `png_memset` procedure, which initializes a block of memory whose size is a function of the PNG width and bitdepth input fields. The `png_memset` procedure is hand coded in assembly language using the SSE2 extensions. This procedure contains a loop that iterates over the block of memory initializing the values in the block. The number of iterations of this loop is a function of the size of the block of memory. The conditional branch that controls the number of iterations is therefore a relevant branch — its condition depends on the PNG width and bitdepth fields. Any input that follows the same path as the seed input through the relevant conditions must therefore have PNG width and bitdepth fields that produce the same number of iterations of the loop as the seed inputs. This additional *blocking constraint* makes it impossible to obtain an input that both 1) triggers the overflow and 2) follows the same path through the relevant branches as the seed input.

In our example, the PNG width field is 280. The number of iterations is 8 and the constraint is width × bitdepth/8 ≤ 1154. The target expression is (width × bitdepth/8) × height (which is rowbytes × height). This value cannot overflow because the maximum value of rowbytes is 1154 and the maximum value of height is 1,000,000 (line 24). These values produce 1154 × 1,000,000 = 1,154,000,000, which is less than 2^{32}.

Goal-Directed Conditional Branch Enforcement: DIODE next starts goal-directed conditional branch enforcement. It initializes the *current constraint* to the target constraint and the *current input* to the initial input (recall that the

initial input was generated to satisfy only the target constraint). It then executes Dillo on the seed input and the current input to find the first (in the program execution order) relevant branch where the seed and current input take different paths. In our example this relevant branch corresponds to the sanity check at function `png_get_uint_31`, line 48 — the seed input satisfies this sanity check, while the current input fails the sanity check (because the generated height is too large).

DIODE therefore adds the branch constraint from the corresponding conditional to the current constraint. Given this new current constraint, Z3 produces a solution that sets `/header/width` to 1632109428L, `/header/bit_depth` to 4, and `/header/height` to 872360950L. The resulting current input fails to generate an overflow because it fails the sanity check at `png_check_IHDR`, line 25.

DIODE adds the branch constraint from the conditional branch that implements the sanity check to the current constraint and obtains a new `/header/width` of 1081489513L and `/header/height` of 732927L. The resulting input file fails to trigger an overflow because it fails the sanity check at `png_check_IHDR`, line 31. After adding the corresponding branch constraint, the solver comes back with `/header/width` 966175L and `/header/height` 484094L. The sanity check at `Png_datainfo_callback`, line 81, which checks for an overly large image size, rejects the resulting current input.

Successful Generation of Overflow-Triggering Input: This sanity check, designed to detect overflows, is itself vulnerable to an overflow — carefully chosen values can overflow the checked value and cause the sanity check to incorrectly accept an input that overflows the target value at line 87. After adding the branch condition from line 81 to the current constraint, the solver comes back with `/header/width` 689853L and `/header/height` 915210L. With these values, the generated input successfully navigates the sanity checks and the blocking checks to trigger the overflow. The resulting out of bounds writes cause Dillo to crash with a SIGSEGV exception.

6.2 Goal-Directed Conditional Branch Enforcement Algorithm

We next present the basic DIODE goal-directed conditional branch enforcement algorithm. We first define a core imperative language and a small-step operational semantics for this language. This semantics defines both concrete and symbolic executions for programs written in the core language. We then use this semantics to present the algorithm.

6.2.1 Core Language

Figure 27 presents the syntax of a core imperative language with variables, arithmetic expressions, boolean expressions, assignments, dynamic memory allocation, memory read/write, conditional statements, while loops, and sequential composition.

$$
\begin{array}{rcll}
x, y & \in & \text{Var} & = \text{PgmVar} \cup \text{InpVar} \\
A, A_1, A_2 & \in & \text{Aexp} & ::= n \mid x \mid \texttt{-}A \mid A_1\, aop\, A_2 \\
B, B_1, B_2 & \in & \text{Bexp} & ::= \texttt{true} \mid \texttt{false} \mid A_1\, cmp\, A_2 \mid \\
 & & & \quad \texttt{!}B \mid B_1 \,\texttt{\&\&}\, B_2 \mid B_1 \,\texttt{||}\, B_2 \\
C, C_1, \cdots, C_n & \in & \text{Stmt} & ::= \texttt{skip} \mid x \texttt{=} A \mid \\
 & & & \quad x \texttt{=} \texttt{alloc(}y\texttt{)} \mid x \texttt{=} y\texttt{[}A\texttt{]} \mid x\texttt{[}A\texttt{]} \texttt{=} y \mid \\
 & & & \quad \texttt{if}\, B\, S_1\, S_2 \mid \texttt{while}\, B\, S \\
S, S_1, S_2 & \in & \text{Seq} & ::= C_1 \texttt{;} \cdots \texttt{;}\, C_n
\end{array}
$$

Figure 27: Syntax

Variables: We divide variables into two classes, PgmVar and InpVar. A *program variable* $\in$ PgmVar is a conventional variable and can store integer values or memory addresses as usual. On the other hand, an *input variable* $\in$ InpVar represents an external input value to a program. DIODE uses input variables to symbolically express how the program computes a target value (such as the size of the allocated memory block) from the input values.

Labels: All program statements have a unique label $\ell \in$ Label. before(C) and after(C) denote the labels before and after the statement C, respectively. In a sequence $S = C_1 \texttt{;} \cdots \texttt{;} C_n$, after($C_i$) = before($C_{i+1}$). We define before($C_1 \texttt{;} \cdots \texttt{;} C_n$)

and after$(C_1; \cdots; C_n)$ as follows:

$$\begin{aligned} \text{before}(C_1; \cdots; C_n) &= \text{before}(C_1) \\ \text{after}(C_1; \cdots; C_n) &= \text{after}(C_n) \end{aligned}$$

$$\rho \vdash n \Rightarrow (n, n) \qquad \frac{x \in \text{PgmVar}}{\rho \vdash x \Rightarrow \rho(x)} \qquad \text{INPVAR}\ \frac{x \in \text{InpVar}}{\rho \vdash x \Rightarrow (\pi_1(\rho(x)), x)}$$

$$\frac{\rho \vdash A \Rightarrow (n, n)}{\rho \vdash -A \Rightarrow (-n, -n)} \qquad \frac{\rho \vdash A \Rightarrow (n, A^t)}{\rho \vdash -A \Rightarrow (-n, -A^t)}$$

$$\frac{\rho \vdash A_1 \Rightarrow (n_1, n_1) \quad \rho \vdash A_2 \Rightarrow (n_2, n_2)}{\rho \vdash A_1 + A_2 \Rightarrow (n_1 + n_2, n_1 + n_2)} \qquad \frac{\rho \vdash A_1 \Rightarrow (n_1, A_1^t) \quad \rho \vdash A_2 \Rightarrow (n_2, n_2)}{\rho \vdash A_1 + A_2 \Rightarrow (n_1 + n_2, A_1^t + n_2)}$$

$$\frac{\rho \vdash A_1 \Rightarrow (n_1, n_1) \quad \rho \vdash A_2 \Rightarrow (n_2, A_2^t)}{\rho \vdash A_1 + A_2 \Rightarrow (n_1 + n_2, n_1 + A_2^t)} \qquad \frac{\rho \vdash A_1 \Rightarrow (n_1, A_1^t) \quad \rho \vdash A_2 \Rightarrow (n_2, A_2^t)}{\rho \vdash A_1 + A_2 \Rightarrow (n_1 + n_2, A_1^t + A_2^t)}$$

Figure 28: Semantics of Arithmetic Expressions

6.2.2 Operational Semantics

The language has three different kinds of values

$$\begin{aligned} n &\in \text{Int} \\ b, b_1, b_2 &\in \text{Bool} = \{\text{true}, \text{false}\} \\ a &\in \text{Addr} \end{aligned}$$

where Int is a set of machine integers of finite bit-width, Bool is the standard set of boolean values, and Addr is an address space with an unbounded number of memory addresses.

An *environment* $\rho \in$ Env is a partial mapping from variables to pairs of values and symbolic values. A *value* $v \in$ Val is either an integer or an memory address. A *symbolic value* $w \in$ SymVal can be a symbolic arithmetic expression, integer, or memory address. We use symbolic values to characterize how values were computed as a function of input variables.

$$\begin{aligned} \rho &\in \text{Env} = \text{Var} \to \text{Val} \times \text{SymVal} \\ v, v_1, v_2 &\in \text{Val} = \text{Int} \cup \text{Addr} \\ w, w_1, w_2 &\in \text{SymVal} = \text{Int} \cup \text{Addr} \cup \text{Aexp} \end{aligned}$$

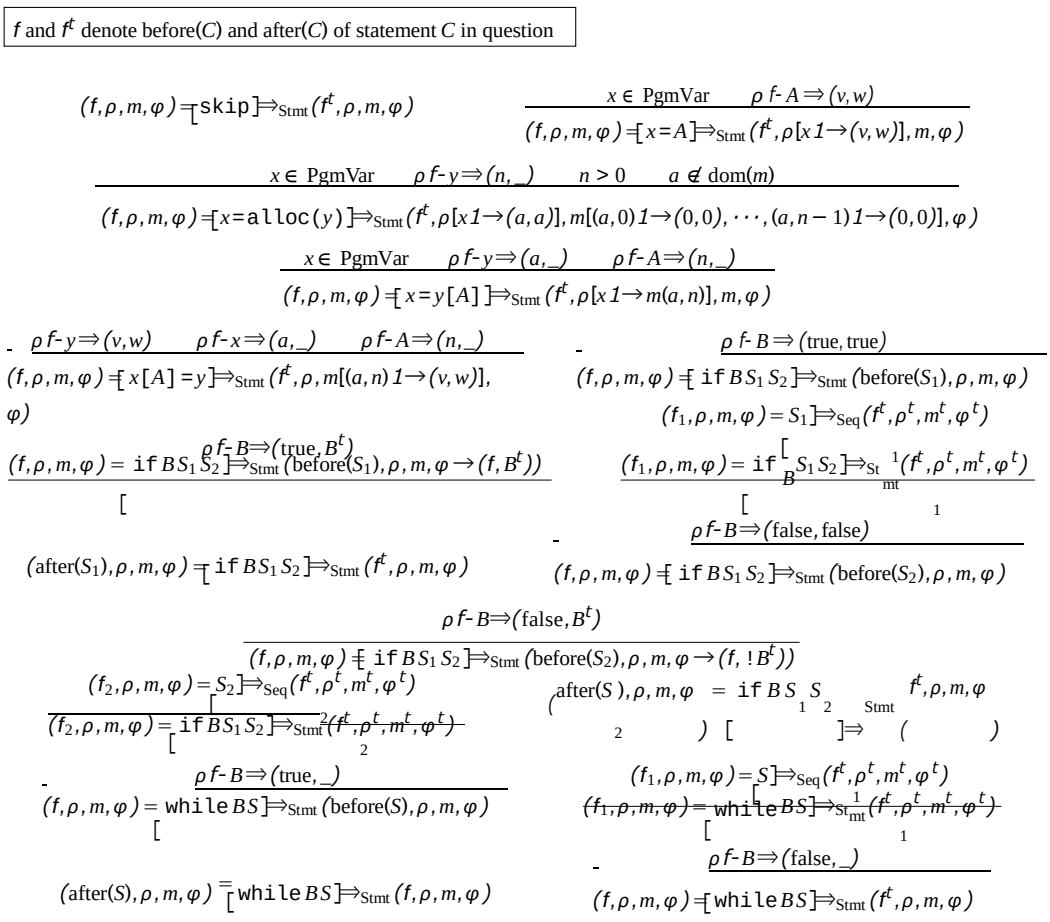


Figure 29: Small-Step Operational Semantics of Statements

Similar to an environment, a *memory* $m \in$ Mem receives a base address and an offset to the base address as its arguments and returns a pair of a value and a symbolic value.

$$m, m_1, m_2 \in \text{Mem} = \text{Addr} \to \text{Offset} \to \text{Val} \times \text{SymVal}$$

A *branch condition* $\varphi \in$ BranchCond is a sequence. Each element (f, B) in this sequence records the symbolic branch condition that determines the path taken at the conditional branch at label f. The elements appear in φ in the program execution order.

$$\varphi \in \text{BranchCond} ::= \varepsilon \mid (f, B) \to \varphi$$

$$\frac{f \text{ is a label in } C_i \qquad (f,\rho,m,\varphi) =[C_i]\Rightarrow_{\text{Stmt}} (f^t,\rho^t,m^t,\varphi^t)}{(f,\rho,m,\varphi) =[C_1;\cdots;C_n]\Rightarrow_{\text{Seq}} (f^t,\rho^t,m^t,\varphi^t)}$$

Figure 30: Small-Step Operational Semantics of Sequences

A *program state* $\sigma = (f,\rho,m,\varphi)$ is composed of the current program point (represented by a label f), an environment ρ, a memory m, and a branch condition φ. At a state (f,ρ,m,φ), the program is about to execute a statement C labelled f (i.e. before$(C) = f$) in the environment ρ and memory m at the program point f reached by taking the path recorded by the conditional branches in the sequence φ.

$$o \in \text{State} = \text{Label} \times \text{Env} \times \text{Mem} \times \text{BranchCond}$$

Expressions: Figure 28 presents the semantics of arithmetic expressions. Each expression evaluates to a pair (v,w), where $v \in$ Val is a concrete value and $w \in$ SymVal is a symbolic expression. The INPVAR rule, for example, defines that the evaluation of an input variable $x \in$ InpVar produces a pair $(\pi_1(\rho(x)),x)$, where $\pi_1(\rho(x))$ is the actual input value and x is the variable that symbolically represents that value. The semantics of boolean expressions is defined in a similar way.

Statement: Figures 29 and 30 present the small-step operational semantics of DIODE's core language. Note that the meaning of f and f^t is slightly different in Figures 29 and 30. In Figure 29, f is the label for the program point before the relevant statement C; f^t is the label for the program point after C. In Figure 30, f and f^t are the labels of some program points within (including before for f and after for f^t) some statement C_i in $C_1;\cdots;C_n$.

6.2.3 Algorithm

Figure 31 presents the DIODE goal-directed conditional branch enforcement algorithm. Given a program S, an initial program state σ, and a target site f, the algorithm first extracts the symbolic target expression B and the observed path φ (from the seed input) for that site (line 1). target$((S,\sigma),f)$ is defined as follows:

$$\text{target}((S,\sigma),f) = \{(\pi_2(\rho(y)),\varphi) \mid (\sigma,(f,\rho,m,\varphi)) \in \tau^*(S)\}$$
$$\text{where } f = \text{before}(x = \texttt{alloc}(y))$$

The function target((S, σ), f) is defined in terms of the reflexive transitive closure τ^* (S) of the transition relation of the program S, which contains all possible
transitions from a starting state to all reachable states.

The algorithm next uses the overflow(B) function to extract the target constraint β (line 2). The overflow(B) function returns a target constraint β such that any input that satisfies the target constraint β will trigger an overflow during the computation of the target expression B.

The algorithm next compresses the path φ to coalesce multiple occurrences of conditional branch constraints of a conditional statement into a single constraint (line 7 and Figure 32). This single constraint is the conjuction of all of the observed branch constraints. The algorithm then extracts the relevant branch constraints (line 8) and performs the goal-directed conditional branch enforcement algorithm (lines 10-16).

The relevant(φ, β) function takes a branch condition φ and a target constraint β as its arguments, and removes conditions that are not relevant to the target constraint β from the branch condition φ. A condition (T, B) in a branch condition is relevant to a target constraint β if the condition B and the target constraint β share the same input variable.

6.2.4 System Design and Implementation

We next discuss how DIODE deals with the many complications that it must overcome to effectively operate on stripped x86 binaries. DIODE consists of approximately 9,000 lines of C (most of this code implements the taint and symbolic expression tracking) and 6,000 lines of Python (the target and branch constraint generation algorithms, code that interfaces with Z3, code that manages the database of relevant experimental results, and a distributed work queue system). We first describe our techniques for target site identification. Second, we introduce the dynamic instrumentation used for target and branch constraint extraction. Third, we discuss how DIODE generates and solves target constraints. Fourth, we discuss how DIODE generates new inputs. Fifth, we discuss the implementation of our goal-directed conditional branch enforcement algorithm. Finally, we discuss how DIODE detects any errors caused by the overflow.

Input : a program S, an initial state σ, a target label f
Output : an input I that triggers an integer overflow at label f

```
1  for (B, φ) in target((S, σ), f) do
2      β ←− overflow(B)
3      if the solver generates an input I that satisfies β then
4          if the input I triggers an overflow at label f then
5              return the input I
6      else continue
7      φ ←− compress(φ)
8      φ ←− relevant(φ, β)
9      φ^t ←− true
10     while true do
11         if the previous input I satisfies φ then break
12         φ^t ←− φ^t ∧ (the first condition in φ that the previous
                   in- put I does not satisfy)
13         if the solver generates an input I that satisfies φ^t ∧ β then
14             if the input I triggers an overflow at label f then
15                 return the input I
16         else break
17 return not found
```

Figure 31: Goal-Directed Conditional Branch Enforcement

6.2.5 Target Site Identification

To extract the set of symbolic target expressions that characterizes how the application computes the target value at critical program sites, DIODE uses a fine-grained dynamic taint analysis built on top of the Valgrind [47] binary analysis framework. Our analysis takes as input a specified taint source, such as a file-name or a network connection, and marks all data read from the taint source as tainted. Each input byte is assigned a unique label and is tracked by the execution monitor as it propagates through the program until it reaches a potential target site (e.g., malloc). To track the data-flow dependencies from source to sink, our analysis instruments arithmetic instructions (e.g., ADD, SUB), data movement instructions (e.g., MOV, PUSH) and logic instructions (e.g., AND,

Parameters: $\varphi \in$ BranchCond

Returns : φ's compressed form $\in$ BranchCond

```
1 Function compress(φ) =
2 begin
3     if φ is ε then
4         return ε
5     else if φ is (f, B) → φ then
6         B ←− B ∧ ( ⋀_{(f,B^t) in φ} B^t )
7         φ ←− filter out all (f, B^t) from φ
8         return (f, B) → compress(φ)
```

Figure 32: Branch Condition Compression

XOR). Using the dynamic taint analysis on the application and a seed input, DIODE generates the set of target sites and relevant input bytes.

6.2.6 Target and Branch Constraint Extraction

Next, DIODE reruns the program with additional instrumentation that enables DIODE to reconstruct the full symbolic target expression. Conceptually, DIODE generates a symbolic record of all calculations that the application performs (Section 6.2). Obviously, attempting to record all calculations would produce an unmanageable volume of information. DIODE reduces the volume of recorded information with the following optimizations:

- **Relevant Input Bytes:** DIODE only records calculations that involve the relevant input bytes. Specifically, DIODE maintains an expression tree of relevant calculations that only tracks calculations that operate on tainted data (i.e., relevant input bytes). This optimization drastically reduces the amount of recorded information.
- **Simplify Expressions:** DIODE further reduces the amount of recorded information by simplifying recorded expressions at runtime. Specifically, DIODE identifies and simplifies resize, move and arithmetic operations. For example, DIODE can convert the following sequence of VEX IR instructions:

```
t15 = Add32(t10, 0x1:I32)
t16 = Add32(t15,0x1:I32)
```

```
t17 = Add32(t16,0x1:I32)
```

that would result in: `Add32(Add32(Add32(t10, 0x1), 0x1),0x1)`

into: `Add32(t10, 0x3)`

To convert relevant input bytes to symbolic representations of the input format, DIODE uses the Hachoir [8] tool to convert byte ranges into input fields (e.g., in the PNG format, bytes 0-3 represent /header/height).

DIODE also uses the recorded information to extract symbolic expressions that characterize how the application computes the values of conditional branch instructions that relevant input bytes directly influence.

6.2.7 Target Constraint Solution

DIODE uses the Z3 SMT solver [33] to obtain new input values that satisfy the target constraint. Note that the generated target constraint is designed to capture any overflow in the evaluation of the expression, including in the evaluation of subexpressions. For example, if $bbp_8 \in \{8, 16, 32\}$, there are no values that cause the following expression to overflow:

$$((width_{16} \times height_{16}) \times 4)/bbp_8) > 2^{32}$$

But there are values that cause the following subexpression to overflow:

$$((width_{16} \times height_{16}) \times 4)) > 2^{32}$$

6.2.8 Test Input Generation

DIODE uses a combination of Hachoir [8] and Peach [15] to generate input files with the values obtained from the SMT solver for the target expression. Together, these tools reconstruct the input file such that it satisfies any checksum calculations or any required field orderings. If DIODE needs to operate on an unknown input format, it also supports a raw-byte option, where modifications are made directly on the input bytes. To deal with any required checksum calculations in raw-byte mode, DIODE can use standard checksum reconstruction techniques [56].

6.2.9 Goal-Directed Branch Enforcement

If a test input that is generated from a target constraint solution fails to trigger an integer overflow error, DIODE turns on instrumentation that records the path taken at all conditional branches that the seed input executes. DIODE uses this instrumentation to find the first conditional branch at which the generated input takes a different path from the seed input. DIODE uses this information to drive the goal-directed branch enforcement algorithm described above (Section 6.2).

6.2.10 Error Detection

We use Valgrind's *memcheck* to detect errors (invalid reads and writes; uninitialized reads and writes) that occur as a result of the overflow. Our automated system therefore does not directly detect the overflow; it only detects the overflow indirectly through its effect on the computation (for our benchmark applications, we manually verify that the generated input actually produces an overflow and generates the reported errors as a result of the overflow). Our automated system first filters any errors that occur during t he execution on the seed input.

6.2.11 Evaluation

We evaluate DIODE on five applications: Dillo 2.1, VLC 08.6h, SwfPlay 0.5.5, CWebP 0.3.1, and ImageMagick 6.5.2. For each application we obtain a seed input, then use DIODE to automatically generate input files that trigger overflows in the applications. We perform all tests on a quad Intel i7 2.2 GHz machine with 8 GB RAM.

6.2.12 Benchmark Selection

The benchmark applications were selected as follows. First, we select applications that process input formats supported by Hachoir [8] and Peach [15]. Second, we filter applications that cannot be successfully processed by DIODE's dynamic instrumentation engine. Third, we select applications that contain at least one known integer overflow vulnerability,

Application	Total Target Sites	DIODE Exposes Overflow	Target Constraint Unsatisfiable	Sanity Checks Prevent Overflow
Dillo 2.1	12	3	1	8
VLC 08.6h	4	4	0	0
SwfPlay 0.5.5	8	3	5	0
CWEBP 0.3.1	7	1	6	0
ImageMagick 6.5.2	9	3	5	1

Table 1: Target Site Classification

6.2.13 Target Site Classification

Table 1 classifies the target sites in our benchmark applications. There is one row for each application. The first column (Application) identifies the application. The second column (Total Target Sites) presents total number of exercised memory allocation sites from the executions on the seed inputs. These sites are the target sites. The third column (DIODE Exposes Overflow) presents the number of sites for which DIODE was able to generate an input that triggered an overflow at the site. The fourth column (Target Constraint Unsatisfiable) presents the number of sites for which the target constraint, by itself, is unsatisfiable. We verified, via a manual inspection, that there is *no* input that will cause an overflow at any of these sites. The fifth column (Sanity Checks Prevent Overflow) presents the number of remaining sites. For all of these remaining sites, we manually verified that the application contains sanity checks that ensure that there is *no* input that triggers an overflow at that site.

Note that, for each target site, either 1) DIODE finds an input that triggers an overflow at that site, or 2) no such input exists. Our analysis indicates that, except for VLC 0.8.6h, whenever DIODE is able to generate an input that triggers an overflow at a given site, the application is missing overflow sanity checks for that site (of course, the applications contain other relevant sanity checks that DIODE must successfully navigate to trigger the overflow). VLC 0.8.6h contains ineffective overflow sanity checks that are designed to protect the application against overflow, but do not, in fact, do so. DIODE is able to generate inputs that successfully evade these checks to trigger overflows at the target sites.

Application	Target	CVE Number	Error Type	Analysis and Discovery Time	Enforced Branches	Target Success Rate
Dillo 2.1	png.c@203	CVE-2009-2294	SIGSEGV/InvalidRead	(42m) 8m	4/35	0/200
Dillo 2.1	fltkimagebuf.cc@39	New	SIGSEGV/InvalidRead	(42m) 7m	5/69	0/200
Dillo 2.1	Image.cxx@741	New	SIGSEGV/InvalidRead	(42m) 7m	4/5779	0/200
VLC 0.8.6h	messages.c@355	New	SIGSEGV/InvalidRead	(6m) 1m	2/117	32/200
VLC 0.8.6h	wav.c@147	CVE-2008-2430	InvalidRead/Write	(6m) 1m	0/62	2/2
VLC 0.8.6h	dec.c@277	New	SIGSEGV/InvalidRead	(6m) 8m	5/291	57/200
VLC 0.8.6h	block.c@54	New	InvalidRead	(6m) 4m	0/151	200/200
SwfPlay 0.5.5	jpeg_rgb_decoder.c@253	New	SIGSEGV/InvalidWrite	(7m) 13m	0/1736	200/200
SwfPlay 0.5.5	jpeg_rgb_decoder.c@257	New	SIGSEGV/InvalidWrite	(7m) 13m	0/1736	200/200
SwfPlay 0.5.5	jpeg.c@192	New	SIGABRT/InvalidWrite	(7m) 1m	0/1012	200/200
CWebP 0.3.1	jpegdec.c@248	New	SIGSEGV/InvalidWrite	(11m) 2s	0/651	155/200
ImageMagick 6.5.2	xwindow.c@5619	CVE-2009-1882	SIGSEGV/InvalidWrite	(6m) 1m	0/2521	200/200
ImageMagick 6.5.2	cache.c@803	New	SIGSEGV/InvalidWrite	(6m) 1m	0/306	199/200
ImageMagick 6.5.2	display.c@4393	New	SIGSEGV/InvalidWrite	(6m) 2m	0/154	200/200

Table 2: Evaluation Summary

6.2.14 Overflow Characteristics

Table 2 summarizes the results for each overflow. The table contains one line for each overflow that DIODE discovers. The first column (Application) identifies the application that contains the overflow. The second column (Target) presents the source code file and line that contains the memory allocation statement for which the overflow occurs. The third column (CVE Number) presents either the CVE number of the overflow (if the overflow was known) or "New" if the overflow was new. We note that all but three of the 14 overflows were new. Four of the 11 new overflows persist in the latest versions of the benchmark applications as of the submission date of this paper. Specifically, the latest versions of CWebP and Display, CWebP 0.4.1 and Display 6.8.9-8, are still vulnerable to error triggering inputs discovered by DIODE. We have notified the developers and are awaiting confirmation.

The fourth column (Error Type) characterizes the effect of the overflow on the application for the first input (that DIODE discovers) that triggers the overflow. In most cases the overflow causes the program to generate a SIGSEGV exception and crash, either from an invalid read or from an invalid write as presented in the table. The remaining two overflows cause the application to perform invalid reads and/or writes that do not crash the application. We detect these invalid reads and writes using the Valgrind memcheck tool [47], which monitors the reads and writes and detects invalid reads and writes. All of the invalid reads or writes occur because the overflow makes the memory block allocated at the

target allocation site too small to contain the data.

The fifth column (Analysis and Discovery Time) presents the initial analysis time required for each application (performed once) and the subsequent time to generate an error input for each bug. Each entry in this column is of the form (A) B, where A is the analysis time and B is the time required to generate the error input.

The sixth column (Enforced Branches) presents the number of relevant conditional branches that DIODE enforced before generating an input that triggered the overflow. Each entry in this column is of the form X/Y, where X is the number of enforced conditional branches and Y is the total number of relevant conditional branches on the path that the seed input takes to the target memory allocation site. We note that the number of enforced conditional branches is small, especially relative to the total number of relevant conditional branches — to discover the overflow, DIODE enforces only between two to five out of the 35 to 5779 total relevant conditional branches. Our manual inspection of the code indicates that all of the enforced branches are sanity checks, but that (apparently) only one of these checks is designed (obviously incorrectly) to detect an overflow (Section 7.2).

6.2.15 Blocking Checks

Recall that DIODE can generate a constraint that requires 1) the computation of the target value to overflow and 2) the input to follow the same path through the relevant conditional branches as the seed input. If this constraint is satisfiable, the solution typically immediately provides an input that will trigger an overflow at the site. Because of blocking checks, this constraint is unsatisfiable for all but two of the sites, specifically SwfPlay 0.5.5 at jpeg.c@192 and CWebP 0.3.1 at jpegdec.c@248.

6.2.16 Inputs That Satisfy Target Constraint Alone

The seventh column (Target Success Rate) presents the results from the experiment in which DIODE generated 200 inputs that satisfied the target constraint by itself (with none of the conditional branch constraints added to the target

constraint passed to the solver). Note that all of these inputs will trigger an overflow at the target memory allocation site if they follow a path that evaluates the target expression at that site. Note also that every discovered input that triggers the overflow is in the set of inputs that satisfy the target constraint alone and therefore could potentially be generated as one of the sampled 200 inputs.

Each entry in the column is of the form X/200, where X is the number of generated inputs that actually trigger the overflow. We note that there is a bimodal distribution — in general, either all or the vast majority of the 200 generated inputs trigger the overflow or none or few of the 200 generated inputs trigger the overflow. This bimodal distribution is correlated with the presence or absence of sanity checks on relevant input values — without sanity checks, all or the vast majority of the generated inputs trigger the overflow. If the application contains sanity checks, the generated inputs are unlikely to pass the sanity checks to trigger the overflow. These data indicate that, if the application contains sanity checks and the input generation strategy does not take these checks into account, the input generation strategy is unlikely to find inputs that trigger an overflow (even when such inputs exist).

For CVE-2008-2430, the target expression is of the form $x + 2$, where x is an input field. The target constraint for this expression has only two solutions (because there are only two values of x that cause the target expression to overflow).

6.2.17 Target and Enforced Branch Success Rate

The eighth column (Target + Enforced Success Rate) presents experimental results for those overflows that DIODE discovered only after enforcing some of the conditional branches. DIODE generated 200 inputs that satisfied the corresponding constraint (i.e., the target constraint plus the constraints that enforced the discovered first flipped branches in Algorithm 31). Each entry in the column is of the form X/200, where X is the number of generated inputs that trigger the overflow (note that we do not run this experiment if the majority of the inputs that satisfy the target constraint alone also trigger the overflow).

We note that, for three of the five overflows, the vast majority of the generated inputs trigger the overflow. For the remaining two overflows, approximately half of the generated inputs trigger the overflow. We attribute this success to DIODE's ability to produce inputs that satisfy the sanity checks while preserving their

flexibility to satisfy the blocking checks and traverse alternate paths through the computation to reach the target memory allocation site and trigger the overflow.

The success of DIODE in generating these overflows also illustrates the difficulty of writing sanity checks that detect inputs that cause overflows — even though Dillo 2.1 and VLC 0.8.6h contain sanity checks, these checks do not detect all inputs that trigger overflows.

7 CodePhage

Horizontal gene transfer is the transfer of genetic material between cells in different organisms. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [25]), virally-mediated gene therapy [38], and the transfer of insect toxin genes from bacteria to fungal symbionts [23]. Because of its ability to directly transfer functionality evolved and refined in one organism into another, horizontal gene transfer is recognized as a significant factor in the development of many forms of life [39].

Like biological organisms, software applications often face challenges and threats from the environment in which they operate. Despite significant software development effort, errors and security vulnerabilities still remain a important concern. Many of these errors are caused by an uncommon case that the developers of one (or more) of the applications did not anticipate. But in many cases, the developers of another application did anticipate the uncommon case and wrote correct code to handle it.

7.1 The Code Phage (CP) Code Transfer System

We present Code Phage (CP), a novel *horizontal code transfer system* that automatically eliminates errors in recipient software applications by finding correct code in donor applications, then transferring that code from the donor into the recipient. The result is a software hybrid that productively combines beneficial code from multiple applications:

- **Donor Selection:** CP starts with an application and two inputs: an input that triggers an error and a seed input that does not trigger the error. Working with a database of applications that can read these inputs, it locates a donor that processes both inputs successfully. The hypothesis is that the donor contains a check, missing in the recipient, that enables it to process the error-triggering input correctly. The goal is to transfer that check from the donor into the recipient (and eliminate the error in the recipient).
- **Candidate Check Discovery:** To identify the check that enables the donor to survive the error-triggering input, CP analyzes the executed conditional branches in the donor to find branches that take different directions for the seed and error-triggering inputs. The hypothesis is that if the check elimi-

nates the error, the seed input will pass the check but the error-triggering input will fail the check (and therefore change the branch direction).

- **Patch Excision:** CP performs an instrumented execution of the donor on the error-triggering input to obtain a symbolic expression tree that expresses the check as a function of the input fields that determine its value. This execution translates the check from the data structures and name space of the donor into an application-independent representation suitable for insertion into another application.
- **Patch Insertion:** CP next uses an instrumented execution of the recipient on the seed input to find *candidate insertion points* at which all of the input fields in the excised check are available as recipient program expressions. At each such point, it is possible to translate the check from the application-independent representation into the data structures and name space of the recipient. This translation, in effect, inserts the excised check into the recipient.
- **Patch Validation:** CP inserts the translated check into the recipient at each candidate insertion point in turn, then attempts to validate the patch. It recompiles the application, uses regression testing to verify that the patch preserves correct behavior on the regression suite, and checks that the patch enables the patched recipient to correctly process the error-triggering input. As available, CP also reruns error detecting tools to generate additional error-triggering inputs, which it then uses to recursively eliminate any residual or newly discovered errors.

  As appropriate, CP can also exploit the semantics of specific classes of errors (such as divide by zero or integer overflow) to perform additional validation steps. For integer overflow errors, for example, CP analyzes the check, the expression that overflows, and other existing checks in the recipient that are relevant to the error to verify that there is no input that 1) satisfies the checks to traverse the exercised path through the program to the overflow and also 2) triggers the overflow.
- **Retry:** If the validation fails, CP tries other candidate insertion points, other candidate checks, and other donors.

If the transferred check detects an input that may trigger the error, it exits the application before the error occurs. The check therefore introduces no new and potentially unpredictable behaviors — it simply narrows the set of inputs that the application decides to process. This narrowing is conceptually similar to transformations that eliminate concurrency errors by narrowing the set of possible

interleavings [45, 37].

7.1.1 Usage Scenarios

Proprietary Donors: The CP donor analysis operates directly on stripped binaries with no need for source code or symbolic information of any kind. CP can therefore use arbitrary binaries, including closed-source proprietary binaries, as donors for other applications. A developer could, for example, reduce development and testing effort by simply omitting checks for illegal inputs, then using CP to automatically harden the application by automatically transferring in checks from more intensively engineered (including closed-source proprietary) applications.

Multilingual Code Transfer: CP supports multilingual code transfer between applications written in different programming languages. Because CP works with binary donors, the current implementation supports arbitrary (compiled) donors. The current CP implementation generates source-level patches in C. It would be straightforward to extend CP to generate patches in other languages. Given appropriate binary patching capability, it would also be straightforward to generate binary patches, including hot patches for running applications.

Multiversion Code Transfer: In addition to transferring checks between independently developed applications, we have also used CP to transfer checks between different versions of the same application. The motivation is to automatically obtain a targeted update that eliminates an error in an older version without the disruption often associated with full upgrade [31].

Divergent Functionality: Even though CP works with applications that process the same inputs, the recipient and donor do not need to implement the same functionality. Many errors occur in code that parses the input, constructs the internal data structures that hold the input, and/or reads the input into those data structures. Even when the applications have different goals and functionality, the fact that they both read the same input is often enough to enable a successful transfer.

Continuous Multiple Application Improvement: CP can work with any source of seed and error-triggering inputs. Its current integration with the DIODE automatic error-discovery system [52] points the way to future systems that combine 1) large libraries of applications, 2) a variety of automatic error discovery tools

(for example, DIODE and BuzzFuzz [35]), and 3) CP along with other automatic error repair tools such as ClearView [50], staged program repair [42], and automatic code fracture and recombination [53]. Continuously running the error-discovery tools across the library of applications, then using horizontal code transfer and other program repair mechanisms to generate repairs delivers an automatic application improvement system that productively leverages the entire global software development enterprise.

Such a system holds out the promise of automatically producing robust software hybrids that incorporate the best code produced anywhere by any mechanism. Given the ability of DIODE and CP to work with stripped binary donors, it is possible to include closed-source software produced by proprietary software development efforts into this continuous application improvement system.

7.1.2 Scope

CP is currently designed to locate and transfer checks, including all computation required to compute the checks, between applications that process the same inputs. The goal is to change the (incorrect) semantics of the recipient so that it rejects inputs that would otherwise trigger the error. The patch validation phase, along with the rejection of unstable insertion points (Section 7.3), is designed to reduce, but not necessarily eliminate, the possibility of rejecting inputs that the recipient could have processed correctly. The excised computation can be, and in practice always is, distributed across multiple system layers and abstraction boundaries within the donor — the excised computation always includes code from multiple system libraries and procedures within the application.

In the current implementation of CP, a set of values sufficient to compute the check must be available in the recipient at one of the granularities at which they are accessed in the excised computation and in one of the same byte orders. It is straightforward to extend the implementation to reassemble values sufficient to compute the check from bits arbitrarily distributed across the address space of the recipient as long such a set of bits is accessible via the name space of the recipient.

CP is currently designed to transfer code that computes a check. But the basic CP transfer techniques are designed to dynamically track, extract, and insert any computation (or computations) that generate any value (or values) in the donor

as long as CP can identify the value(s). The two critical questions are identifying the value(s) in the donor and the insertion point(s) in the recipient. CP automates this identification for checks in the donor that eliminate errors in the recipient.

7.1.3 Experimental Results

We evaluate CP on 10 errors in 7 recipient applications (JasPer 1.9 [10], gif2tiff 4.0.3 [11], CWebP 0.31 [3], Dillo 2.1 [4], swfplay 0.55 [18], Display 6.5.2-8 [9], and Wireshark-1.4.14 [22]). The donor applications are FEH-2.9.3 [5], mtpaint 3.4 [13], ViewNoir 1.4 [20], gnash 0.8.11 [7], OpenJpeg 1.5.2 [14], Display 6.5.2-9 [9], and Wireshark-1.8.6 [22]. CP was able to successfully generate patches that eliminated the errors, in five cases demonstrating the ability to transfer patches from multiple donors (see Section 7.4).

For all of the applications except Wireshark-1.4.14 (which uses Wireshark 1.8), CP successfully excises code from an independently developed alien donor and successfully implants the code into the recipient. The ability of CP to translate the check from the donor name space and data structures into the name space and data structures of the recipient is critical to the success of many transfers. Wireshark-1.4.14 demonstrates the ability of CP to deliver targeted updates that eliminate specific errors while leaving the behavior and functionality of the recipient otherwise intact.

7.1.4 Contributions

This paper makes the following contributions:

- **Basic Concept:** CP automatically eliminates software errors by identifying and transferring correct code from donor applications into incorrect recipient applications. In this way CP can automatically harness the combined knowledge and labor invested across multiple software systems to improve each application.
  To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.
- **Name Translation:** One of the major challenges in code transfer is translating the names of values from the name space of the donor into the name

space of the recipient. CP shows how to use instrumented executions of the donor and recipient to meet this name translation challenge.

- **Data Structure Translation:** Another major code transfer challenge is translating between different data representations. CP shows how to use instrumented executions and data structure traversals to meet this challenge — it takes code that accesses values stored in the data structures of the donor and produces code that accesses values stored in the data structures of the recipient.
- **Donor Code Identification:** It presents a mechanism to identify correct code in donor applications for transfer into recipient applications. CP uses two instrumented executions of the donor to automatically identify the correct code to transfer into the recipient: one on the seed input and one on the error-triggering input (which the donor, but not the recipient, can successfully process). A comparison of the paths that these two inputs take through the donor enables CP to isolate a single check (present in the donor but missing in the recipient) that eliminates the error.
- **Insertion Point Identification:** CP automatically identifies appropriate check insertion points within the recipient at which 1) the values needed to express the transferred check computation are available as valid program expressions in the name space of the recipient and 2) the transferred check will not affect observed computations unrelated to the error.
- **Experimental Results:** We present experimental results that characterize the ability of CP to eliminate ten otherwise fatal errors in seven recipient applications by transferring correct code from seven donor applications. For all of the 10 possible donor/recipient pairs, CP was able to obtain a successful validated transfer that eliminated the error.

7.2 Example

We next present an example that illustrates how CP automatically patches an integer overflow error in CWebP, the Google conversion program for the WepP image format.

Figure 33 presents (simplified) CWebP source code that contains an integer overflow error. CWebP uses the libjpeg library to read JPG images before converting them to the CWebP format. It uses the `ReadJPEG` function to parse the JPG files. There is a potential overflow at line 9, where CWebP calculates

```
1  int ReadJPEG(...) {
2      ...
3      width    = dinfo.output_width;
4      height   = dinfo.output_height;
5      stride   = dinfo.output_width *
6                      dinfo.output_components *
7                      sizeof(*rgb);
8      /* the overflow error */
9      rgb = (uint8_t*)malloc(stride * height);
10     if (rgb == NULL) {
11         goto End;
12     }
13     ...
14 }
```

Figure 33: (Simplified) CWebP Overflow Error

the size of the allocated image as stride * height, where stride is: width * output_components * sizeof(rgb).

On a 32-bit machine, inputs with large width and height fields can cause the image buffer size calculation at line 9 to overflow. In this case CWebP allocates an image buffer that is smaller than required and eventually writes beyond the end of the allocated buffer.

Error Discovery: In our example, CP works with seed and error-triggering inputs identified by the DIODE integer-overflow discovery tool, which performs a directed search on the input space to discover inputs that trigger integer overflow errors at memory allocation sites [52]. In the error-triggering input in our example, the JPG height field is 62848 and the width field is 23200.

Donor Selection: CP next searches a database of applications that process JPG files to find candidate donor applications that successfully process both the seed and the error-triggering inputs. In our example, CP determines that the FEH image viewer application processes both inputs successfully.

Candidate Check Discovery: CP next runs an instrumented version of the FEH donor application on the two inputs. At each conditional branch that is influenced by the relevant input field values (in this case the JPG height and width fields), it records the direction taken at the branch and a symbolic expression for the value of the branch condition. The free variables in these expressions represent the values of input fields.

CP operates under the hypothesis that one of the FEH branch conditions implements a check designed to detect inputs that trigger the overflow. Under this

```
1  # define IMAGE_DIMENSIONS_OK(w, h) \
2      ( ((w) > 0) && ((h) > 0) && \
3        ((unsigned long long)(w) * \
4         (unsigned long long)(h) <= (1ULL << 29) - 1) )
5
6  char load(...) {
7      int     w, h;
8      struct jpeg_decompress_struct cinfo;
9      struct ImLib_JPEG_error_mgr jerr;
10     FILE    *f;
11     ...
12     if (...) {
13         ...
14         im->w = w = cinfo.output_width;
15         im->h = h = cinfo.output_height;
16         /* Candidate check condition */
17         if ((cinfo.rec_outbuf_height > 16) ||
18                 (cinfo.output_components <= 0) ||
19                 !IMAGE_DIMENSIONS_OK(w, h))
20         {
21             // Clean up and quit
22             ...
23             return 0;
24         }
25     }
26 }
```

Figure 34: (Simplified) FEH Overflow Check

hypothesis, the seed input and error-triggering inputs take different directions at this branch (because the error-triggering input would satisfy the check and the seed input would not). CP therefore considers the check for each branch at which the seed and error-triggering inputs take different directions to be a *candidate check*.

In our example, CP discovers a candidate check in the imlib library that FEH uses to load and process JPG files. Figure 34 presents (simplified) source code for this check.[3] The macro IMAGE_DIMENSIONS_OK (defined on lines 1-4, invoked on line 19), performs an overflow check on the computation of output_width * output_height. This check enables FEH to detect and correctly process the error-triggering input without overflow.

Candidate Check Excision: The FEH check is expressed in terms of the FEH data structures. The next step is to translate the check from this form into an

[3] Because CP operates on binaries, information about the source code for the donor patch is, in general, not available. So that we can present the FEH source code for the check in our example, we used the symbolic debugging information in FEH to manually locate the source code for the check.

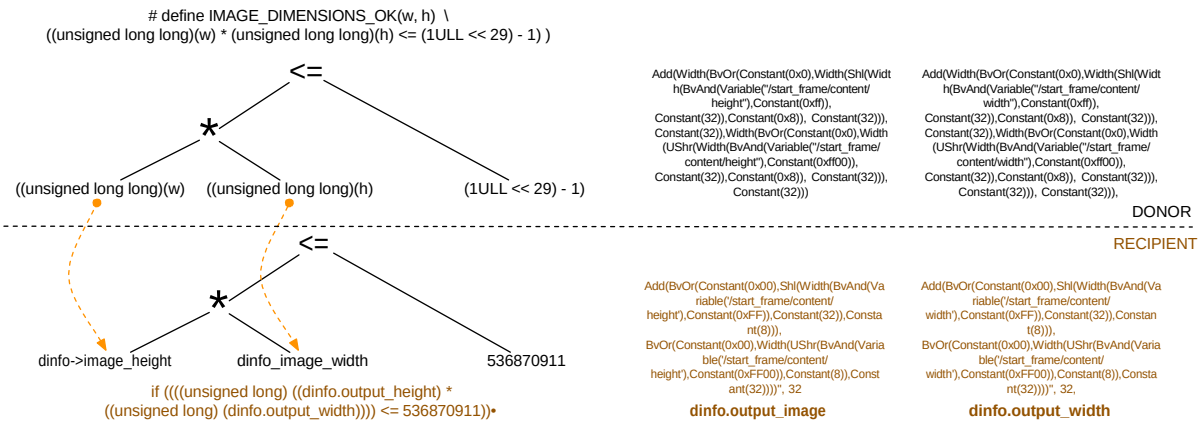


Figure 35: Patch Transfer

application-independent form that expresses the check as a function of the input bytes that determine its value. This translation uses an instrumented execution of the donor to dynamically track the flow of input bytes through program. CP uses this instrumentation to obtain symbolic expressions, in terms of the input bytes, for relevant expressions that the application computes. In our example the translated application-independent symbolic expression for the check is:

```
ULessEqual(32,Shrink(32,Mul(64,Shrink(32,Div(32,BvOr(64,Shl(64,
ToSize(64,SShr(32,Sub(32,Add(32,Constant(8),Shl(32,Add(32,Shl
(32,ToSize(32,BvAnd(16,HachField(16,'/start_frame/content/height'),
Constant(0xFF))),Constant(8)),ToSize(32,UShr(32,BvAnd(16,HachField(16,
'/start_frame/content/height'),Constant(0xFF00)),Constant(8)))),
Constant(3))),Constant(1)),Constant(31))),Constant(32)),ToSize(64,
Sub(32,Add(32,Constant(8),Shl(32,Add(32,Shl(32,ToSize(32,BvAnd(16,
HachField(16,'/start_frame/content/height'),Constant(0xFF))),Constant(8)),
ToSize(32,UShr(32,BvAnd(16,HachField(16,'/start_frame/content/height'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)))),Constant(8))),
Shrink(32,Div(32,BvOr(64,Shl(64,ToSize(64,SShr(32,Sub(32,Add(32,
Constant(8),Shl(32,Add(32,Shl(32,ToSize(32,BvAnd(16,HachField(16,
'/start_frame/content/width'),Constant(0xFF))),Constant(8)),
ToSize(32,UShr(32,BvAnd(16,HachField(16,'/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)),
Constant(31))),Constant(32)),ToSize(64,Sub(32,Add(32,Constant(8),
Shl(32,Add(32,Shl(32,ToSize(32,BvAnd(16,HachField(16,
'/start_frame/content/width'),Constant(0xFF))),Constant(8)),
ToSize(32,UShr(32,BvAnd(16,HachField(16,'/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)))),
Constant(8))))),Constant(536870911))
```

There are two primary reasons for the complexity of this excised check. First, it correctly captures how FEH manipulates the input fields to convert from big-endian (in the input file) to little-endian (in the FEH application) representation. The excised check correctly captures the shifts and masks that are performed as part of this conversion. Second, FEH casts the 16-bit input fields to unsigned long long integers before it performs the overflow check. The excised check

properly reflects these operand length manipulations.

Patch Transfer: The next step is to insert the check into the recipient CWebP application. There are two related challenges: 1) finding a successful insertion point for the check and 2) translating the check from the application-independent representation into the data representation of the recipient CWebP application. Note that this translation must find CWebP data structures that contain the relevant input field values and express the check in terms of these data structures.

Candidate Patch Insertion Point Identification: CP runs CWebP (the recipient) on the seed input. At each function the CP instrumentation records the input fields that the function reads. CP identifies program points at which the function has read all of the input fields as potential patch insertion points. In our example, CP recognizes that the ReadJPEG function has read both the input JPG width and height fields after line 4 in Figure 33. It therefore identifies the point after this statement as a candidate insertion point. The next step is to use the variables and data structures available at this point to express the check.

Patch Translation: To translate the patch into the recipient, CP first finds the relevant input fields as stored in the variables and data structures of the recipient. It then determines how to use these fields to express the check.

To find the values, CP uses the debugging information from the recipient binary to identify the local and global variables available at that candidate insertion point. Using these variables as roots, it traverses the data structures to find memory locations that store relevant input fields or values computed from relevant inputs fields and constants. As part of this traversal it also records expressions (in the name space of the recipient) that evaluate to each of the input fields or input field expressions. In our example CP determines that dinfo.height contains the JPG height input field and dinfo.width contains the JPG width input field.

The next step is to use the extracted recipient expressions to express the extracted check in the name space of the recipient. CP recursively processes the application-independent expression tree to find subtrees that always evaluate to the same value as one of the extracted recipient expressions. CP uses an SMT solver to determine this equivalence (see Section 7.3). In our example, CP produces the following translated check, which it inserts after line 4 in Figure 33:

```
if (!((unsigned long long)dinfo.output_height *
     (unsigned long long)dinfo.output_width)<=536870911)) {
    exit(-1);
}
```

Note that CP was able to successfully convert the complex application-independent excised check into this simple form — the SMT solver detects that CWebP and FEH, even though developed independently, perform semantically equivalent endianess conversions, shifts, and masks on the input fields. CP therefore realizes that the input fields are available in the same format in both the CWebP and FEH internal data structures, enabling CP to generate a simple patch that accesses the CWebP data structures directly with no complex format conversion. The generated patch evaluates the check and, if the input fails the check, exits the application. The rationale is to exit the application before the integer overflow (and any ensuing errors or vulnerabilities) can occur.

Multiple Patch Insertion Points: For CWebP, CP identifies 38 candidate insertion points. 2 of these points are *unstable* — in some executions of the point, the generated expressions reference values other than the desired JPEG `width` and `height` input fields. To avoid perturbing computations not related to the error, CP filters out these unstable points. CP then sorts the remaining generated patches by size and attempts to validate the patches in that order. In our example the above patch is the first patch that CP tries (and this patch validates).

Patch Validation: Finally, CP rebuilds CWebP, which now includes the generated patch, and subjects the patch to a number of tests. First, it ensures the compilation process finished correctly. Second, it executes the patched version of CWebP on the error-triggering input and checks that the input no longer triggers the error (CP runs CWebP under Valgrind memcheck to detect any errors that do not manifest in crashes). Third, it runs a regression test that compares the output of the patched application to the output of the original application, on a regression suite of inputs that the application is known to process correctly. Fourth, CP runs the patched version of the application through the DIODE error discovery tool to determine if DIODE can generate new error-triggering inputs. In our example DIODE finds no new error-trigging inputs — if it had, CP would have rerun the entire patch discovery and generation process, patching the discovered errors, until DIODE discovered no new errors. The end result, in this example, is a version of CWebP that contains a check that completely eliminates the integer overflow error.

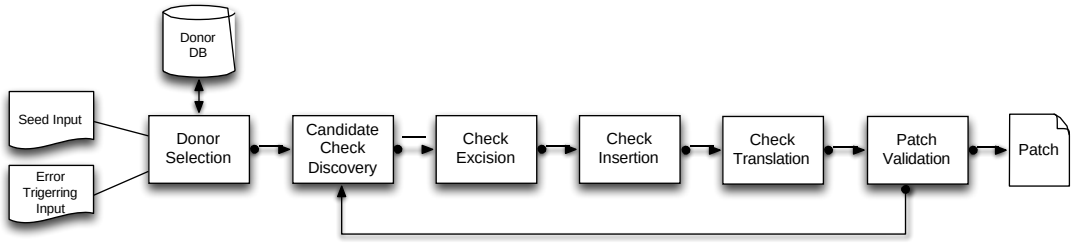


Figure 36: High-level overview of CP's components

7.3 Design and Implementation

We next discuss how CP deals with the many technical issues it must overcome to successfully transfer code between applications. CP consists of approximately 10,000 lines of C (most of this code implements the taint and symbolic expression tracking) and 4,000 lines of Python (code for rewriting donor expressions into expressions that can be inserted into the recipient, code that generates patches from the bitvector representation, code that interfaces with Z3, and the code that manages the database of relevant experimental results). Figure 36 presents an overview of the CP components.

Donor Selection For each input format, CP works with a set of applications that process that format. Given seed and error-triggering inputs, CP considers applications that can successfully process both inputs as potential donors. Open source repositories such as github can be a rich source of independently developed applications that process the same input formats. Different versions, releases, or variants of the same application can also be good sources of patches either for regression errors introduced during maintenance or to obtain targeted updates for specific errors. Our set of benchmark donors includes both sources of applications (Section 7.4).

Candidate Check Discovery and Excision To extract candidate checks from donor applications, CP implements a fine-grained dynamic taint analysis built on top of the Valgrind [47] binary analysis framework. Our analysis takes as input a specified taint source, such as a file or a network connection, and marks all data read from the taint source as tainted. Each input byte is assigned a unique label and is tracked by the execution monitor as it propagates through the application.

Our analysis instruments arithmetic instructions (e.g., ADD, SUB), data movement instructions (e.g., MOV, PUSH), and logic instructions (e.g., AND, XOR). It also supports additional instrumentation to reconstruct the full symbolic expression of each computed value, which records how the application computes the value from input bytes and constants.

CP can optionally work with only a specified subset of the input bytes. We call this subset the *relevant bytes*. Working with properly identified relevant bytes can often improve the efficiency of the analysis without hampering its ability to find successful patches (because only a subset of the bytes are relevant to the patch). In our experiments, CP identifies the relevant bytes as those input fields that differ between the seed and error-triggering inputs.

CP uses Hachoir [8] to convert byte ranges into symbolic input fields. If Hachoir does not support a particular input format or is otherwise unable to perform this conversion, CP also supports a raw mode in which all input bytes are represented as offsets. Raw mode is effective, for example, for closely related inputs generated by standard error-finding tools [52, 35, 55, 15].

Identify Candidate Checks: CP runs the dynamic taint analysis on the donor application twice, once with the seed input and once with the error-triggering input. For each execution, CP extracts the executed conditional branch instructions and records which direction each execution of the branch takes. After filtering out branches that are not affected by the relevant bytes, branches that take different directions are the *candidate branches*. CP proceeds under the assumption that the condition associated with one of the candidate branches implements the desired check. Starting with the first (in the program execution order) candidate branch, CP attempts to transfer each check in turn until a transferred check successfully validates.

Check Excision: To obtain the application-independent form of the check, CP reruns the application with additional instrumentation that enables CP to reconstruct the full symbolic expression tree for the candidate check. This expression tree records how the donor application computes the condition of the candidate check from the input byte values and constants. Conceptually, CP generates a symbolic record of all calculations that the application performs. To reduce the volume of recorded information, CP only builds expression trees for calculations that involve the relevant input bytes. This optimization substantially reduces the volume of generated data.

A key challenge in transferring code between applications is translating between the different data representations in the donor and recipient. Translating the check into a symbolic expression over the input bytes performs the first half of this translation — it translates the check out of the naming environment and data structures of the donor into an application-independent representation.

Bit Manipulation Optimizations: As the symbolic expressions are recorded during the instrumented execution of the donor, CP applies several optimizations that reduce the size of the generated expressions. Among the most important of these are optimizations that simplify expressions generated by bit manipulation operations (such as shifts) that extract, align, or combine operands of subsequent computations. Because such bit manipulation operations occur frequently (for example, when the application extracts pieces of data read from the input or because of SSE optimizations) in donor binaries, the rules significantly reduce the size and complexity of the extracted symbolic expressions.

Figure 37 presents several rewrite rules that CP applies to simplify the symbolic expressions that such operations generate. The first two rules simplify symbolic expressions that extract the bottom or top 8-bit byte, respectively, of a 16 bit value. Here Shl(8,*E*) represents an 8-bit left shift of the 16 bit value *E*; ShrinkH(8,Shl(8,*E*)) converts the resulting 16 bit value into an 8 bit value by extracting the top byte. One important consequence of these rules is that, by eliminating discarded bytes from the symbolic representation, they can disentangle bytes from adjacent input fields that were read into the same word as part of the input process.

Note that the rules require the operand of the shift to be represented symbolically as a concatenation of two 8-bit bytes (the operand *E* must be of the form $[b_1, b_2]$, where b_1 and b_2 are independent bytes). Potential other representations that may appear as an operand include unified 16-bit values produced by addition or subtraction operations. CP does not further optimize the representation of bit manipulation operations involving such unified operands as there is no straightforward way to disentangle the two bytes of the unified operand.

The last two rules simplify symbolic expressions that start with a 16-bit value composed of two 8-bit bytes, shift one of the bytes out of the value, then or another byte into the position vacated by the shift. Here BvOrH(b_1, Shr(8,*E*)) bitwise ors b_1 into the top byte of the 16 bit value produced by Shr(8,*E*). The result is a new 16 bit value. Once again, one of the benefits of these rules is that they can eliminate bytes that would otherwise entangle unrelated input fields that

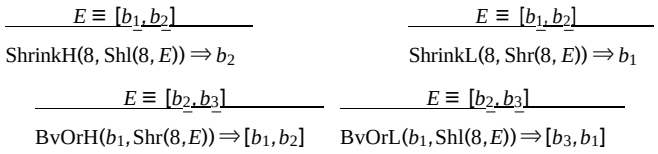


Figure 37: CP Rewrite Rules for Bit Manipulation Operations

appear adjacent in the input. Like the first two rules, the last two rules require the initial 16-bit value to be represented symbolically as a concatenation of two 8-bit values.

CP also implements similar rules for other combinations of operand sizes. Specifically, there are similar rules for expressions that represent results of bit manipulation operations involving combinations of 8, 16, 32, and 64 bit values.

Check Insertion To transfer the candidate check to an insertion point in the recipient application, CP rewrites the check to access the input field values as stored in variables and data structures available in the recipient.

Candidate Insertion Points: The first step is to find *candidate insertion points* – program points at which a set of values computed from all of the input bytes in the symbolic check expression are available as program expressions in the recipient. CP runs an instrumented version of the recipient that tracks the flow of the relevant input bytes through the application. Whenever the recipient evaluates an expression that involves the relevant input bytes, CP records the symbolic expression for the computed value. This symbolic expression records how the recipient application computes the value as a function of the input bytes and constants. Using these collected symbolic expressions, CP finds functions that access a set of values computed from all of the input bytes in the check. It then finds points within these functions at which the function has accessed all of these values. These points are the set of candidate insertion points.

Unstable Points: In general, the application may execute a candidate insertion point multiple times, potentially accessing different input bytes or even different values not derived from the input bytes on different executions. Candidate insertion points in multipurpose code such as libraries, for example, may execute with different values when invoked from different parts of the computation. To minimize the risk that the inserted check may affect a computation not related to the

error, CP filters out all points that access different values on different executions (we call these points *unstable* points). The goal is choose the insertion point so that the patch performs the check only when it is relevant to the error.

Paths to Relevant Values: CP next attempts to express the extracted symbolic check in terms of the available variables and data structures at the remaining stable candidate insertion points. Given a candidate point, CP uses the debugging information to find the set V of local and global variables available at that point. Starting with these variables as roots, it then uses the debugging information to traverse the data structures to find *relevant values* (values computed from relevant fields and constants) stored in the data structures. As part of the traversal it computes the data structure traversal paths that lead to these relevant values.

Figure 38 presents the traversal algorithm. Starting from a given variable or data structure traversal path, the algorithm computes names that lead to reachable relevant values. Each name has the form (p, E). Here p is a path through the reachable data structures. Each path p starts at a variable v, then identifies a sequence of pointer dereferences and data structure field accesses that reaches the relevant value. The symbolic expression E records how the program computed the value from relevant input bytes.

For each variable $v \in V$, CP invokes the traverse algorithm and merges the resulting sets of names. The algorithm recursively traverses the data structures of the recipient program based on type signatures from the debugging information. At line 15, it uses the debugging information to determine the type of the path p. At line 16, it queries the symbolic tracking analysis results to obtain the corresponding symbolic expression for the traversed path p.

Check Translation: The next step is to rewrite the application-independent form of the check to use the variables and data structures of the recipient. Figure 39 presents the CP expression rewrite algorithm. The algorithm takes as input a symbolic expression E and a set of names *Names* produced by the traversal algorithm in Figure 38. It then uses the *Names* to translate E to use the available variables and data structures at the candidate insertion point in the recipient. E may take one of four possible forms: 1) an input field, 2) a constant c, 3) a unary operation expression $(unaryop, E)$, or 4) a binary operation expression $(binop, E_1, E_2)$.

The algorithm first uses an SMT solver to try to find a single value in the recipient with the same value as the expression E. In practice, CP is often able to

find single recipient values that are equivalent to very complex expressions *E* — many of these symbolic expressions include complex shift and mask operations that are also performed by the recipient as it reads the input. Otherwise the algorithm decomposes the expression and attempts to rewrite each subexpression recursively (lines 13-15 for expressions with unary operations, lines 16-19 for expressions with binary operations). Constants (line 20) translate directly.

CP implements two optimizations that reduce the number of solver invocations: 1) if two symbolic expressions depend on different sets of input bytes, CP does not invoke the solver and 2) CP caches all queries to the SMT solver so that it can retrieve results from the cache for future duplicate queries. Together, these two optimizations produce an order of magnitude reduction in the translation times.

There are two ways for the Rewrite algorithm to fail. First, it does not attempt to rearrange or reorder input bits as stored in the recipient data structures to match the groups of input bits as accessed by the application-independent representation of the check. So all of the required input bits may be available in the recipient but not stored as a contiguous block in the order accessed by the check. Second, it is possible for the function to access a value required to compute the check, then overwrite the value before it reaches the insertion point. In this case the value may be unavailable at the insertion point even though it was previously accessed by the enclosing function.

If CP successfully constructs the new condition, CP generates a *candidate patch* as an if statement inserted at the insertion point. In the current implementation, CP transforms the constructed bitvector condition into a C expression as the if condition (appropriately generating any casts, shifts, and masks required to preserve the semantics of the transferred check). If the condition is satisfied, the patch exits the application with an `exit(-1)`.

Patch Validation CP first recompiles the patched recipient application. It then executes the patched application on the bug-triggering input to verify that the patch successfully eliminates the error for that input. CP also runs the patched build on a set of regression suite inputs to validate that the patch does not break the core functionality of the application. As appropriate, CP may also test other error-triggering inputs or run additional error-finding tools (such as DIODE) to determine if the patch leaves any residual errors behind. If so, CP recursively

```
Parameters:
  p: A data structure path.
Subroutines:
  Type(p): The type of the path p.
  Fields(t): If t is a struct type, the set of fields in t.
  Addr(p): The address (at runtime) for the path p.
  Expr(a): The symbolic expression for the value
    stored in the address a.
  Visited(a): A boolean that tracks whether the address
was already processed to avoid infinite recursion.
Returns:
  A set of path, symbolic expression pairs.

Traverse(p) {
  T ← Type(p)
  E ← Expr(Addr(p))
  if (Visited(Addr(p))) return ∅
  else if (T is Pointer) return Traverse("(*"+p+")")
  else if (T is Struct)
    Names ← ∅
    for f in Fields(T)
      Names ← Names ∪ Traverse(p+"."+f)
    return Names
  else if (E /= NIL) return {(p, E)}
  return ∅
}
```

Figure 38: CP Data Structure Traversal Algorithm

```
Parameters:
  E: A symbolic expression over input values.
  Names: A set of available names.
Subroutines:
  SolverEquiv(E1, E2): Query the SMT solver to determine
    whether expressions E1 and E2 are equivalent.
Return:
  Rewritten expression of E or NIL if failed

Rewrite(E, Names) {
  for (p, E^t) in Names
    if (SolverEquiv(E, E^t)) return p
  if (E is of the form (unaryop, E1))
    E1^t ← Rewrite(E1, Names)
    if (E1^t /= NIL) return (unaryop, E1^t)
  else if (E is of the form (binop, E1, E2))
    E1^t ← Rewrite(E1, Names)
    E2^t ← Rewrite(E2, Names)
    if (E1 /= NIL and E2 /= NIL) return (binop, E1^t, E2^t)
  else if (E is Constant c) return c
  return NIL
}
```

Figure 39: CP Rewrite Algorithm

attempts to find and transfer patches that eliminate the residual errors.

7.4 Experimental Results

Recipient	Target	Donor	Generation Time	# Relevant Branches	# Flipped Branches	# Used Checks	# Candidate Insertion Pts	Check Size
CWebP 0.3.1	jpegdec.c:248	feh-2.9.3	4m	157	5	1	38 - 2 - 31 = 5	57 → 4
CWebP 0.3.1	jpegdec.c:248	mtpaint-3.40	4m	94	5	1	38 - 2 - 30 = 6	28 → 2
CWebP 0.3.1	jpegdec.c:248	viewnior-1.4	1m	137	1	1	38 - 2 - 31 = 5	111 → 12
Dillo 2.1	png.c@203	mtpaint-3.40	3m	29	[1,1]	2	16 - 1 - 8 = 7 16 - 1 - 9 = 6	[(18 → 1),(18 → 1)]
Dillo 2.1	png.c@203	feh-2.9.3	3m	120	[4,1]	2	16 - 1 - 9 = 6 16 - 1 - 9 = 6	[(76 → 8), (37 → 3)]
Dillo 2.1	png.c@203	viewnior-1.4	18m	117	1	1	16 - 1 - 9 = 6	79 → 12
Dillo 2.1	fltkimagebuf.cc@39	mtpaint-3.40	13m	29	[1,1]	2	22 - 1 - 10 = 11 22 - 1 - 11 = 10	[(18 → 1),(18 → 1)]
Dillo 2.1	fltkimagebuf.cc@39	feh-2.9.3	2m	120	4	1	22 - 1 - 11 = 10	76 → 9
Dillo 2.1	fltkimagebuf.cc@39	viewnior-1.4	9m	117	1	1	22 - 1 - 11 = 10	79 → 12
Display 6.5.2	xwindow.c@5619	viewnior-1.4	4m	142	6	1	74 - 5 - 60 = 9	55 → 14
Display 6.5.2	xwindow.c@5619	feh-2.9.3	4m	147	6	1	74 - 7 - 58 = 9	17 → 4
Display 6.5.2	display.c@4393	viewnior-1.4	4m	142	6	1	49 - 2 - 45 = 2	55 → 14
Display 6.5.2	display.c@4393	feh-2.9.3	4m	147	6	1	49 - 2 - 45 = 2	17 → 4
SwfPlay 0.5.5	jpeg_rgb_decoder.c@253	gnash	12m	264	7	1	43 - 3 - 35 = 5	53 → 12
SwfPlay 0.5.5	jpeg.c@192	gnash	18m	264	[1,1,3,3]	4	38 - 2 - 34 = 2 38 - 2 - 34 = 2 38 - 0 - 37 = 1 38 - 0 - 37 = 1	[(5 →1),(5 →1), (4 →1),(3 →1)]
JasPer 1.9	jpg_dec.c:492	OpenJpeg 1.5.2	1m	63	19	1	18 - 1 - 16 = 1	188 → 3
gif2tiff 4.0.3	gif2tiff.c:355	Display 6.5.2-9	9m	9	2	1	2 - 1- 0 = 1	3 → 3
Wireshark 1.4.14	packet-dcp-etsi.c:258	Wireshark 1.8.6	4m	101	2	1	40 - 5 - 15 = 20	6 → 2

Figure 40: Summary of CP Experimental Results

We evaluate CP on three classes of errors — out of bounds access, integer overflow, and divide by zero errors. The two out of bounds access errors occur in JasPer 1.9 [10] and gif2tiff 4.0.3 [11] and are triggered by JPEG2K (JasPer) and gif (gif2tiff) images. OpenJPEG [14] and Display 6.5.2-9 [9] are the donors. We use standard fuzzing techniques to obtain the seed and error-triggering inputs.

The seven integer overflow errors occur in four applications: CWebP 0.31 [3], Dillo 2.1 [4], swfplay 0.55 [18], and Display 6.5.2-8 [9]. Two of these errors were listed in the CVE database; one was first discovered by BuzzFuzz [35]; the other four were, to the best of our knowledge, first discovered by DIODE [52]. The errors are triggered by JPG image files (CWebP), PNG image files (Dillo), SWF video files (swfplay), and TIFF image files (Display). The donor applications include FEH-2.9.3 [5], mtpaint 3.4 [13], ViewNoir 1.4 [20], and 0.8.11 [7]. We use DIODE to obtain the seed and error-triggering inputs.

The two divide by zero errors occur in Wireshark-1.4.14 [22] and are triggered by degenerate network packets with zero size fields. Wireshark-1.8.6 is the donor — in this scenario the goal is to obtain a targeted update that eliminates

the error without the potential disruption of a full update to a later version. Starting with an error-triggering input from the corresponding CVE report, we used standard techniques to obtain a corresponding seed input that did not trigger the error.

We obtained integer overflow errors from the DIODE project [52]. The buffer overflow errors are reported as security vulnerabilities in the CVE database (CVE-2012-3352,CVE-2013-4231). We selected donor applications by collecting applications that successfully process the seed and error-triggering inputs. We further filter any applications that use the same underlying library (and version) to process inputs (e.g., we select only one donor application that uses libjpeg to process jpeg images). For every class of errors, we try all combinations of recipient-donor pairs that can process the same inputs.

Results Summary: Figure 40 summarizes the results of these experiments. There is a row in the table for each combination of error and donor. The first column (Recipient) identifies the recipient application that contains the error. The second column (Target) identifies the source code file and line where the vulnerability occurs. The third column (Donor) identifies the donor application. The fourth column (Patch Time) presents the amount of time that CP required to generate the patch.

The fifth column (Relevant Branches) presents the number of branches that depend on relevant values. The sixth column (Flipped Branches) presents the number of branches that take different directions for the seed and error-triggering inputs. Several entries are of the form $[X_1, \ldots, X_n]$. These entries correspond to errors with multiple error-triggering inputs. The first patch eliminates the error for the first input but there is a residual error. Recursive CP executions transfer patches to eliminate each remaining residual error, with an error eliminated per patch transfer. In all cases the final sequence of patches completely eliminates the exposed errors. For all four cases with multiple patches DIODE, running on the previously patched version, automatically generates the additional error-triggering inputs. The seventh column (Used Checks) presents the number of checks that CP transferred to eliminate the error. In all of our experiments, the transferred checks came from the first (in the execution order) flipped branch.

The eighth column (Candidate Insertion Points) contains entries of the form $X - Y - Z = W$. Here X is the number of candidate insertion points, Y is the number of unstable points (CP filters these points), Z is the number of insertion points at which CP was unable to translate the patch (see Section 7.3), and W is

the number of points at which CP is able to insert a successfully translated patch. The ninth column (Check Size) contains entries of the form $X \rightarrow Y$. Here X is the number of operations in the excised application-independent representation of the check. Y is the number of operations in the translated check as it is inserted into the recipient. We attribute the significant size reduction to the ability of the CP Rewrite algorithm (Figure 39) to recognize complex expressions that are semantically equivalent. The typical scenario is that CP recognizes that a complex application-independent expression containing shifts and masks from (for example) the endianess conversion is equivalent to a single variable or data structure field in the recipient.

We next discuss several specific patches in more detail (see Section 7.2 for a detailed example that illustrates how CP corrects an integer overflow error).

7.4.1 JasPer 1.9

JasPer 1.9 is an open-source image viewing and image processing utility. It is specifically known for its implementation of the JPEG-2000 standard. JPEG-2000 images may be composed of multiple tiles, with the number of tiles specified by a 16 bit field in the input file. JasPer contains an off-by-one error in the code that processes JPEG-2000 tiles. When JasPer processes the tiles, it includes code that is designed to check that the number of tiles actually present in the image is less than or equal to the number specified in the input file. Unfortunately, the check was miscoded — at jpc_dec.c:492, JasPer checks if the number of the current tile is greater than (>) the specified number of tiles. The correct check is a greater than or equal to (>=) check. The result is that JasPer can write tile data beyond the end of the buffer allocated to hold that data.

The following correct check appears in OpenJPEG 1.5.2 at j2k.c:1394:[4]

```
if ((tileno < 0) || (tileno >= (cp->tw * cp->th))) { ... }
```

CP automatically locates the compiled version of this correct check in the OpenJPEG binary and correctly transfers the check into JasPer at jpc_dec.c:492 as:

```
if (!(!(dec->numtiles <= sot->tileno))) { exit(-1); }
```

[4] CP does not have access to the OpenJPEG 1.5.2 source code — it instead transfers the check directly from the compiled binary. For presentation purposes, we used the debugging information to manually locate this check in the OpenJPEG source code.

To generate this check, CP had to map `tileno` in OpenJPEG 1.5.2 to `dec->numtiles` in JasPer and recognize that `cp->tw * cp->th` in OpenJPEG 1.5.2 has the same value as `sot->tileno` in JasPer. This patch highlights CP's data structure translation capabilities and its ability to recognize different expressions in different applications that produce the same value. We note that the OpenJPEG `tileno < 0` check is redundant — other constraints in both OpenJPEG and JasPer ensure that `tileno` and `dec->numtiles` are always nonnegative.

7.4.2 gif2tiff

gif2tiff is a utility in the libtiff-4.0.3 library which converts gif images to the tif format. gif2tiff is vulnerable to a buffer overflow attack when processing gif images. gif2tiff iterates over the size of the LZW code size, which under the gif specification should be limited to a size of 12. Without a check to constrain the code size to 12, the loop over the code size in `gif2tif.c:355` can be forced to overwrite over a set of statically allocated buffers.

CP successfully created a patch for this error using ImageMagick-6.5.2-9 as the donor. The transfered check appears in ImageMagick-6.5.2-9 as:

```
#define MaximumLZWBits  12
if (data_size > MaximumLZWBits)
    ThrowBinaryException(CorruptImageError,
            "CorruptImage",image.filename);
```

This check was translated into the following patch for gif2tiff (`gif2tiff.c:357`) as:

```
if (!(datasize <= 12)) {exit(-1);}
```

The check correctly enforces the gif specification that the code size should have a maximum size of 12 and protects gif2tiff from the buffer overflow vulnerability.

7.4.3 Wireshark

Wireshark is a popular open-source packet analyzer. It is used for a variety of networking tasks such as network analysis, network troubleshooting and protocol development. Wireshark 1.4.14 contains a divide by zero error at packet-dcp-etsi.c:276 in code that processes DCP ETSI packets.

The following check, which appears in a later version of Wireshark (1.8.6) and checks that the length of the packet payload is not zero before attempting to further process the packet, eliminates this error:

```
if (real_len) ...
```

Recognizing that `real_len` and `plen` contain the same input fields (the different names reflect the substantial reengineering between the two versions), CP inserts the check into Wireshark 1.4.14 at packet-dcp-etsi.c:258 as:

```
if (!(!(plen == 0))) { exit(-1); }
```

Empirically, returning zero as the result of divide by zero errors often enables the application to continue to execute productively [44]. We therefore implemented an alternate strategy that returns 0 if the check fires rather than exiting. Our results and manual analysis indicate that this strategy delivers correct continued execution for both of the Wireshark divide by zero errors.

7.4.4 Discussion

The patches we present above are, in general, representative of the remaining patches (our CP technical report presents these remaining patches [54]). Like the JasPer patch, 10 of the remaining 18 patches access the stored field values via pointers. This fact highlights the critical role that the CP data structure traversal and rewrite algorithms play in enabling the data structure translations required for successful transfers. As the numbers in Figure 40 indicate, the CP rewrite algorithm is effective at generating compact readable patches — like the patches we present above, they are all expressible in at most several lines of code.

Our manual evaluation of the patches indicates that 1) they all completely eliminate the target error and 2) they do not affect computations unrelated to the error. We attribute this success to three factors: 1) the developers of the donor applications were able to write code that correctly handled the case responsible for the error in the recipient, 2) CP was able to locate and transfer the check that handles this case, and 3) eliminating unstable points is an effective way to filter out the many points that appear in multipurpose library code. The result is focused patches that fire only when necessary to eliminate the target error.

The results also highlight several aspects of CP's techniques. Most of the applications contain more than 100 checks that involve relevant input fields. The

ability of CP to find the single check (within these more than 100 checks) that eliminates the error highlights the effectiveness of CP's check identification technique (which uses flipped branches to isolate the relevant check). CP's ability to find effective patch insertion points among the many potential source code locations highlights the effectiveness of CP's insertion point location algorithm.

All of the transfers involve naming and/or data structure translations. In some cases the translation could be accomplished via a simple variable renaming (if the source code for the donor was available, which it may not be). In other cases there is a more significant data structure translation that involves finding values stored in different structures or accessed via pointers. Even though the application-independent representation of the checks is typically quite complex, CP's Rewrite algorithm is very effective at finding small recipient representations of the check.

Given that programs often deploy different data representations, any general code transfer system requires some data structure translation technique. CP's technique, which is based on representing values as functions of the input bytes, then traversing the data structures to find desired values, would be equally effective for any approach that can establish a correspondence between executions of the donor and recipient.

CP's current data structure translation technique is effective at translating (potentially quite complex) computations that can be expressed as single expressions. Already this technique enables CP to eliminate significant errors in real-world applications. Generalizing CP to support expressions with simple conditionals would be relatively straightforward — augmenting CP's data structure translation technique with a symbolic execution of the two branches would suffice. An effective loop body identification and generalization technique would enable CP to support loops.

8 Conclusion

Modern software projects contain so many defects, and the cost of correcting defects remains so large, that projects typically ship with a long list of known but uncorrected defects.

The CIDER project researched techniques to automate the process of discovering, neutralizing and repairing software bugs and vulnerabilities. As part of this goal, we build components of a continuous automatic improvement system that can automatically search for errors and generate patches that repair the encountered errors. By removing the human from the loop, patch generation time can be reduced, patch robustness improved, leading to fewer unpatched systems.

Our experimental results show that we have the building blocks for creating continuous automatic improvement systems.

References

[1] Amazon mechanical turk. https://www.mturk.com/mturk/welcome.

[2] Common vulnerabilities and exposures. http://cve.mitre.org/.

[3] Cwebp. https://developers.google.com/speed/webp/docs/cwebp.

[4] Dillo. http://www.dillo.org/.

[5] Feh - a fast and light Imlib2-based image viewer. http://feh.finalrewind.org/.

[6] Gimp. http://www.gimp.org/.

[7] Gnu gnash. https://www.gnu.org/software/gnash/.

[8] Hachoir. http://bitbucket.org/haypo/hachoir/wiki/Home.

[9] Imagemagick. http://www.imagemagick.org/script/index.php.

[10] The jasper project home page. http://www.ece.uvic.ca/~frodo/jasper/.

[11] Libtiff. http://www.remotesensing.org/libtiff/.

[12] The LLVM compiler infrastructure. http://www.llvm.org/.

[13] mtpaint. http://mtpaint.sourceforge.net/.

[14] Openjpeg. http://www.openjpeg.org.

[15] Peach fuzzing platform. http://peachfuzzer.com/.

[16] Picasa. http://picasa.google.com/.

[17] SPIKE fuzzing platform. http://www.immunitysec.com/resources-freesoftware.shtml.

[18] Swfdec. http://swfdec.freedesktop.org/wiki/.

[19] Swftools. http://www.swftools.org/.

[20] Viewnoir - the elegant image viewer. http://xsisqox.github.io/Viewnior/.

[21] VLC media player. http://www.videolan.org/.

[22] Wireshark. https://www.wireshark.org/.

[23] Karen Ambrose, Albrecht Koppenhofer, and Faith Belanger. Horizontal gene transfer of a bacterial insect toxin gene into the epichloe fungal symbionts of grasses. *Scientific Reports*, 4, July 2014.

[24] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10. USENIX Association, 2010.

[25] Miriam Barlow. What Antimicrobial Resistance Has Taught Us About Horizontal Gene Transfer. *Methods in Molecular Biology*, 532:397–411, 2009.

[26] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[27] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[28] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07. ACM, 2007.

[29] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07. ACM, 2007.

[30] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05. ACM, 2005.

[31] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 221–236. ACM, 2007.

[32] Weidong Cui, Marcus Peinado, and Helen J. Wang. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 760–770. IEEE Press, 2012.

[35] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[36] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[37] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

[38] Mark A. Kay, Joseph C. Glorioso, and Luigi Naldini. Viral vectors for gene therapy: the art of turning infectious agents into vehicles of therapeutics. *Nat Med*, 7(1):33–40, January 2001.

[39] Patrick J. Keeling and Jeffrey D. Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 9(8), 8 2008.

[40] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming*

language design and implementation, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[41] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. MIT-CSAIL-TR-2011-044.

[42] Fan Long and Martin Rinard. Staged Program Repair in SPR. Technical Report MIT-CSAIL-TR-2015-008, 2015.

[43] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 439–452. ACM, 2014.

[44] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via error shepherding. In *Proceedings of the 35th ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '14'. ACM, 2014.

[45] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, 2010.

[46] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, pages 67–82. USENIX Association, 2009.

[47] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. PLDI '07, 2007.

[48] James Newsome, David Brumley, and Dawn Xiaodong Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[49] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[50] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst,

and Martin Rinard. Automatically patching errors in deployed software. SOSP '09. ACM, 2009.

[51] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07. ACM, 2007.

[52] Stelios Sidiroglou, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic integer overflow discovery using goal-directed conditional branch enforcement. In *MIT CSAIL Technical Report*, 2014.

[53] Stelios Sidiroglou-Douskos, Eli Davis, and Martin Rinard. Horizontal code transfer via program fracture and recombination. Technical Report MIT-CSAIL-TR-2015-012, 2015.

[54] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by multi-application code transfer. Technical Report MIT-CSAIL-TR-2015-013, 2015.

[55] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007.

[56] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10)*, 2010.

[57] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M.F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 163–177. USENIX Association, 2012.

[58] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: black-box exploit detection and signature generation. CCS '06. ACM, 2006.

[59] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. PLDI 2015.